The best programming guide money can buy!

NEW!

CODING MADE SIMPLE

Learn how to program the easy and fun way

helloworld.py - /home/jonni/python - Geany
File Edit Search View Document Project Build Tools Help
Symbols Documents
helloworld.py
Functions
loop [7]
main [4]
1.
2.
3
4
5
Status
Compiler
XPS

```
#!/usr/bin/env python3
# Python intro 2016

def main():
    print('Hello world')

def loop():
    for j in range(5):
        print(j)

if __name__ == '__main__':
    main()
```

```
17:59:13: This is Geany 1.23.1.
17:59:13: Failed to load one or more session files.
17:59:13: File /home/jonni/python/helloworld.py opened(1).
18:00:58: File /home/jonni/python/helloworld.py saved.
18:01:47: File /home/jonni/python/helloworld.py saved.
```

line: 2 / 13 col: 14 sel: 0 INS TAB mode: Unix (LF) encoding: UTF-8 filetype: Python scope: unknown

LEARN HOW TO…

- Master Python basics
- Start coding on your Raspberry Pi
- Complete exciting projects

Future

CODING
MADE SIMPLE

Learn how to program the easy and fun way

CODING
MADE SIMPLE

EDITORIAL TEAM

ART EDITOR
Efrain Hernandez-Mendoza

EDITOR
Neil Mohr

EDITOR-IN-CHIEF
Graham Barlow

CONTRIBUTORS
Les Pounder, Mayank Sharma, Graham Morrison, Ben Everard, Mike Saunders, Jonni Bidwell

IMAGES
ThinkStock, Future Photo Studio

MANAGEMENT

EDITORIAL DIRECTOR
Paul Newman

GROUP ART DIRECTOR
Steve Gotobed

MARKETING

MARKETING MANAGER
Richard Stephens

PRINT & PRODUCTION

PRODUCTION MANAGER
Mark Constance

PRODUCTION CONTROLLER
Vivienne Calvert

CIRCULATION

TRADE MARKETING MANAGER
Juliette Winyard
Phone +44(0)7551 150984

LICENSING

SENIOR LICENSING & SYNDICATION MANAGER
Matt Ellis
matt.ellis@futurenet.com
Phone +44(0)1225 442244

SUBSCRIPTIONS

UK reader order line & enquiries: 0844 848 2852
Overseas reader order line & enquiries: +44 (0)1604 251045
Online enquiries: www.myfavouritemagazines.co.uk

PRINTED IN THE UK BY

William Gibbons on behalf of Future.
Distributed in the UK by Seymour Distribution Ltd,
2 East Poultry Avenue, London EC1A 9PT. Phone: 020 7429 4000

Future Publishing Limited
Quay House, The Ambury, Bath, BA1 1UA, UK www.futureplc.com
www.myfavouritemagazines.co.uk
Phone +44 (0)1225 442244   Fax +44 (0)1225 732275

Future Publishing Limited (company number 2008885) is registered in England and Wales. Registered office: Registered office: Quay House, The Ambury, Bath, BA1 1UA. All information contained in this publication is for information only and is, as far as we are aware, correct at the time of going to press. Future cannot accept any responsibility for errors or inaccuracies in such information. You are advised to contact manufacturers and retailers directly with regard to the price and other details of products or services referred to in this publication. Apps and websites mentioned in this publication are not under our control. We are not responsible for their contents or any changes or updates to them.

If you submit unsolicited material to us, you automatically grant Future a licence to publish your submission in whole or in part in all editions of the magazine, including licensed editions worldwide and in any physical or digital format throughout the world. Any material you submit is sent at your risk and, although every care is taken, neither Future nor its employees, agents or subcontractors shall be liable for loss or damage.

Future

Future is an award-winning international media group and leading digital business. We reach more than 57 million international consumers a month and create world-class content and advertising solutions for passionate consumers online, on tablet & smartphone and in print.

Future plc is a public company quoted on the London Stock Exchange (symbol: FUTR).
www.futureplc.com

Chief executive Zillah Byng-Thorne
Non-executive chairman Peter Allen
Chief financial officer Penny Ladkin-Brand
Managing director, Magazines Joe McEvoy
Tel +44 (0)1225 442 244

We encourage you to recycle this magazine, either through your usual household recyclable waste collection service or at recycling site.

recycle
When you have finished with this magazine please recycle it.

FSC
www.fsc.org
MIX
Paper from responsible sources
FSC® C007184

We are committed to using only magazine paper which is derived from well managed, certified forestry and chlorine-free manufacture. Future Publishing and its paper suppliers have been independently certified in accordance with the rules of the FSC (Forest Stewardship Council).

Welcome!

Learning to code will change the way you think about the world. It's an exciting journey!

Coding is the new cool. With the internet driving a new world of information exchange, business start-ups and online gaming, coders have suddenly become the gatekeepers to these new realms. Combine the huge interest in learning how to create and control these worlds with the surge of cool devices, such as the Raspberry Pi, and you've got a head of technological steam the world hasn't seen since the coding craze of the early 1980s.

Back then, it was more Code Britannia than Cool Britannia – jump forward to today, and Britain is again firmly at the heart of a web and coding revolution. A Brit invented the web, a Brit designed the Raspberry Pi, and Britain is firmly pushing coding to the fore of education. So no matter if you're looking to relive those heady '80s coding days or are a newbie looking to take your first steps into the coding world, you hold in your hands the ideal guide to start coding. Thanks to a new generation of open free software, we all can access operating systems, development tools, compilers and the programming languages needed to create professional programs, apps and tools. We'll show you how to get up and running with a Linux system, then access everything you need freely online.

Coding is easy, exciting and fun. We'll explain the basics, move on to more advanced topics, explain how you can use the Raspberry Pi, and provide you with exciting and easy-to-follow projects. So what are you waiting for? Get coding!

Neil Mohr, Editor

The MADE SIMPLE Manifesto

Made Simple books are designed to get you up and running quickly with a new piece of hardware or software. We won't bombard you with jargon or gloss over basic principles, but we will…

✓ Explain everything in plain English so you can tackle your new device or software with confidence and really get to know how to use it

✓ Break instructions down into easy-to-follow steps so you won't be left scratching your head over what to do next

✓ Help you discover exciting new things to do and try – exploring new technology should be fun and our guides are designed to make the learning journey as enjoyable as possible for you

✓ Teach you new skills you can take with you through your life and apply at home or even in the workplace

✓ Make it easy for you to take our advice everywhere with you by giving you a free digital edition of this book you can download and take with you on your tablet, smartphone or laptop – see page 146 for more details on this offer

How are we doing? Email **techbookseditor@futurenet.com** and let us know if we've lived up to our promises!

CODING MADE SIMPLE

Contents

Get coding

Coding basics

Further coding

Raspberry Pi

Coding projects

CODING

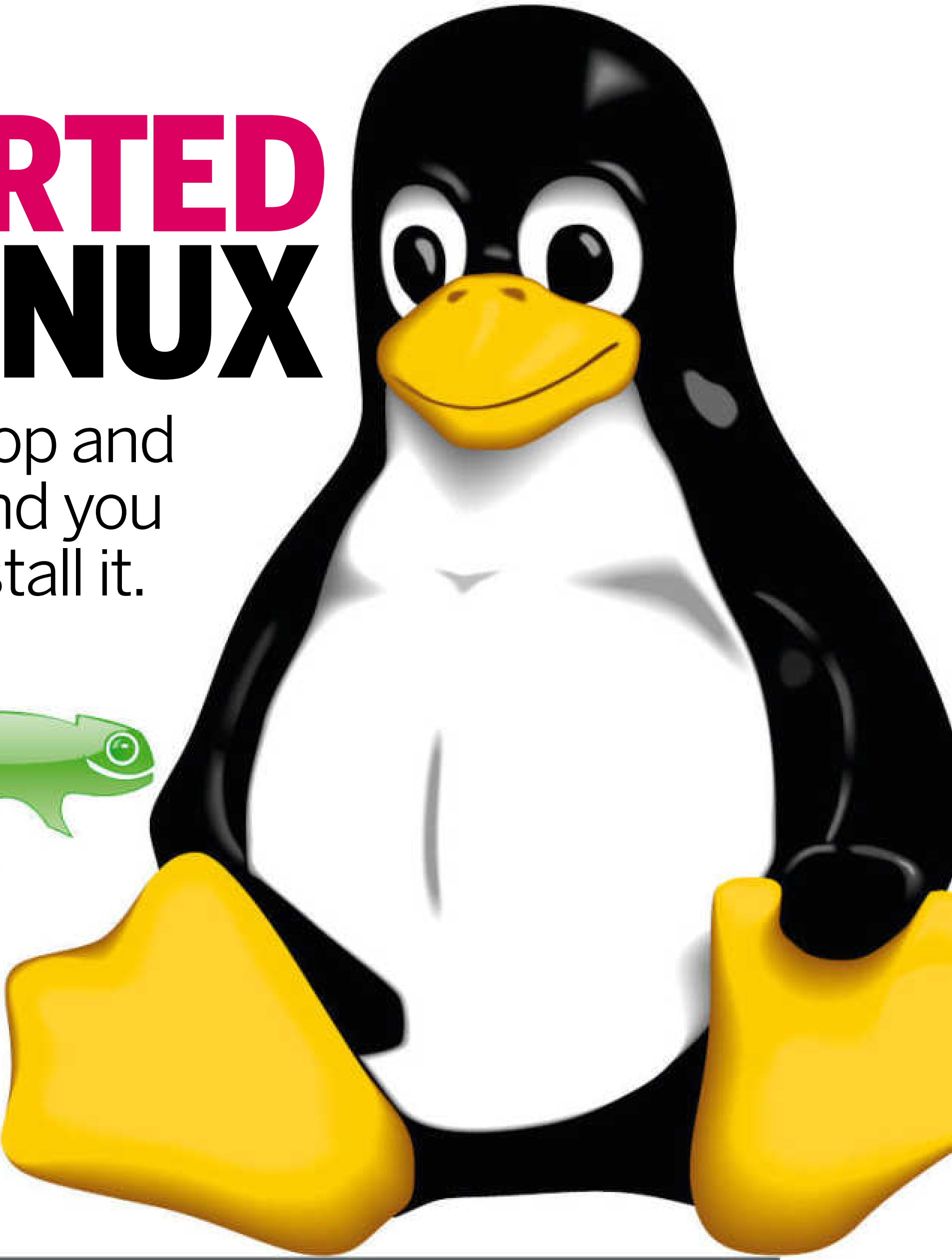
MADE SIMPLE

Get coding!

Everything you need to start coding today

GET STARTED WITH LINUX

It runs on most desktop and laptop PCs, it's free and you don't even need to install it. Let's look at Linux!



As you read through this *Coding Made Simple* bookazine, you'll notice that most of the screens don't look like Microsoft Windows or Apple Mac OS X. There's a good reason for that: they're not. Just as there are different car manufacturers or makes of TV, there's more than one operating system that can run your PC or Mac. It's just that Linux happens to be free because it's developed by thousands of coders around the world.

> "It's not scary, it won't damage your PC and you don't even have to install anything."

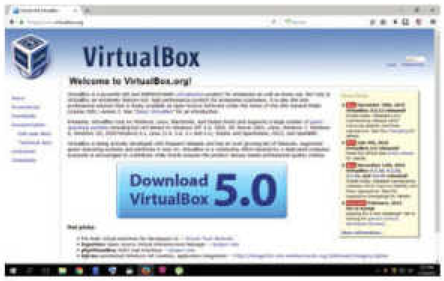
All the coding projects in this bookazine are based on someone running Linux on their desktop. You don't have to – the code works on Windows or Mac OS X – but Linux comes with many languages built in or ready to install from a central server. No scouting round dodgy sites; just run a command or fire up a software centre to get what you need.

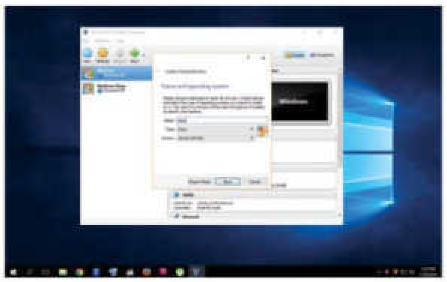
That's the beauty of Linux – it's built by geeks for geeks to do geeky things. Once you're used to its slightly different interface and way of working, you'll find how easy it really is to get along with. We recommend that you use a version (known as a distro) called Linux Mint, and we're going to look at the ways you can get hold of Mint and then get it up and running on your PC.

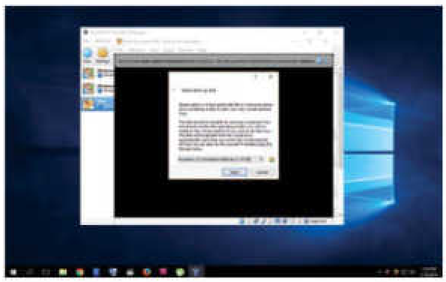
It's not scary, it won't damage your PC and you don't even have to install anything if you don't want to. If you currently have a Windows PC, there are three options: run Linux in VirtualBox, run it off a DVD or USB drive, or install it on your PC dual-booting with Windows.

We're only going to look at the first two as they're the least likely to cause any damage. The walkthrough opposite explains how to run Mint within VirtualBox on top of Windows (or Mac OS X), while the next walkthrough shows you how to create a live image that you can boot and run from either a DVD or a USB flash drive.

VirtualBox







1 Get VirtualBox

Head to **www.virtualbox.org** and download Virtual Box 5 for your operating system, be that Windows or OS X. Install it and be aware you'll need around 20GB of spare drive space to store the virtual OS file. You also need the Mint ISO file from **www.linuxmint.com**. Once installed, start VirtualBox, click New Machine, choose Linux and call it 'Mint'.

2 Create a machine

Choose Ubuntu and the bits should match the ISO you downloaded. Click Next. Under Memory, we recommend 2048, but if you have an 8GB PC, 4096 is best. You can leave all the rest as default settings, apart from the dynamic hard drive size. The default is 8GB, but we suggest 32GB just in case. Finish and click Start to get going.

3 Start virtual Mint

A prompt asks for a disc. Locate the Mint ISO file you downloaded and click Start. Linux Mint starts and, once loaded, you're free to try it out or use the Install icon to properly install Mint to the virtual machine. For extended use, in the virtual machine's settings under Display, you should enable 3D acceleration and allocate 16MB of memory.

If you've never tried Linux, we think you'll be surprised at how easy it is to use, and how good a development platform it makes. You should also know that Linux runs the majority of the internet, from its core servers to websites, alongside powering 97% of the world's supercomputers, and being used widely in science and industry. So it's not such a bad thing to understand, after all.

No matter which tutorial you follow, you'll need a copy of the Linux Mint Cinnamon distro. Head to **www.linuxmint.com/download.php** and download the 32-bit build, unless you know your system is 64-bit, in which case get that. If you have an older or slower machine, opt for the MATE edition, which is less graphically demanding.

Up and running

A big reason why Linux makes such a good development platform is that it was created by developers. In many ways, the key weakness of Linux has been its lack of ease of use (Mint is an exception to that rule) but then, when it's designed and created by developers, ease of use is going to come at the bottom of the to-do list. The upshot of this history is that there's a wealth of the most advanced tools freely available on all Linux platforms. As long as you're willing to seek out help yourself and contribute back to the community, you'll find a very welcoming and rich development ecosystem waiting for you.

Getting apps

With Windows, in the past you've been used to getting programs by downloading them from here, there and everywhere. More recently, the introduction of the Windows Store has at least centralised where software comes from and removed the worry of getting infected by viruses and malware. The fact is that with Linux, the key way of getting new tools and programs has always been from a central repository of software, protected and maintained by the distro's creators.

This is one of the reasons why Linux has remained so secure (it's not infallible) but people downloading dodgy software is a key way that machines become infected. With Linux Mint, there's the Software Center, which gives you access to hundreds of programs and all the programming development tools you

»

Top tip

What's a distro?

Unlike Windows and Mac OS X, because Linux is free software, anyone can take it and effectively create their own OS to distribute. In the Linux world, these are called distros for short, and there are literally hundreds out there – not all good, not all maintained, but hundreds nonetheless.

The GNU of GNU/Linux

The GNU project (GNU stands for 'GNU's Not Unix') predates Linux by several years. It had created most of the basic tools needed by a computer of the early 1980s – compilers, text editors, file and directory manipulation commands, and much more – but did not have a usable kernel (some would say its kernel, GNU Hurd, is still not that usable). When Linus Torvalds started tinkering with his small project in 1991, he had a kernel without the tools to run on it. The two were put together and GNU/Linux was born – an operating system using the Linux kernel and the GNU toolset. It is not only the programs in **/bin** and **/usr/bin** that come from GNU; glibc is the core C library used in Linux and it also comes from GNU. So just about any time you do anything on your computer – every time you type a command or click an icon – GNU software is being run at some level.

No wonder the GNU die-hards get upset when we refer to our operating system as Linux and not GNU/Linux. It is worth mentioning that no one really denies the importance of the GNU aspect; calling the OS Linux rather than GNU/Linux has far more to do with convenience and laziness than politics – the full title is just too cumbersome.

Easy ways to run Linux

If you're not a big Linux user, then you'll probably not want to destroy your existing Windows or Mac system, and we don't blame you. The truth is, you don't need to – Linux is flexible enough that it can be run in a number of ways beside, on top of or alongside most other operating systems, and on most types of hardware – from virtual versions to versions running off spare USB drives, DVDs or on low-cost hardware such as the Raspberry Pi. The standard way, once you've got your hands on the ISO, is to burn it to a DVD. When you first turn on a PC, you can usually get it to boot from alternative media by pressing F11/F12, or hold down C on the Mac – some PCs boot from a suitable optical disc by default.

Another option is to install VirtualBox from **www.virtualbox.org** (see previous page). Install and run this – it looks complex, but creating a virtual PC is pretty easy if you stick to the default settings. The main stumbling block is ensuring under Storage that you add the ISO file to the virtual optical drive. There are more options available, including writing the ISO file to a suitable USB thumb drive and, following a similar boot process as discussed above, running Linux from this. To get this to work, you need to use a write tool such as UNetbootin from **http://unetbootin.github.io**.

For the brave-hearted, you can also install Linux on your system directly. Most versions of Linux create space alongside Windows and make a dual-boot system.

» could wish for. It's not why we're here but you can also download Valve's Steam gaming client and take advantage of over 1,700 Linux games it has to offer.

Drivers

We're not really here to talk about using Linux in every detail but there are a few standard questions that often crop up when people move over from Windows. One key one is where are all the drivers? The cool thing with Linux is that, on the whole, there's no need to worry about drivers – they're built into the Linux kernel. That isn't to say you can't add drivers, but they're generally not required. There are a couple of exceptions: certain more obscure laptop wireless cards can cause issues, while if you want maximum 3D gaming performance, you need to install the dedicated graphics driver from your card's manufacturer.

The Terminal

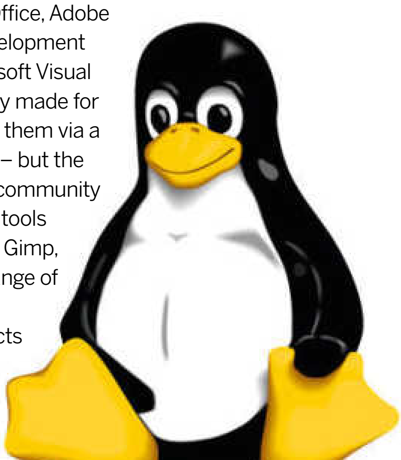
If you've heard of Linux, then one area you might fear or just wonder about is a thing called the Terminal. This can be called a number of different things, depending on the system, such as the command line, command prompt or command-line interface. It is a direct interface to the operating system and all of its tools, which you access through text commands. Going back to the early days of computers and Linux, as computers were so much slower, there weren't any graphical interfaces, so computers were controlled entirely through text commands.

Linux was developed originally in this type of environment, so all of its core tools are based on Terminal use. Many of them – or all the ones you'll care about – do have graphical interfaces these days. The fact is, the Terminal remains an efficient way of controlling Linux, and when it comes to troubleshooting, it offers a consistent set of tools and interfaces to resolve problems.

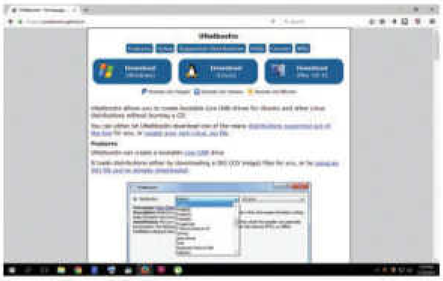
You don't need to know anything about the Terminal to use Linux on a day-to-day basis, but it's good to know that it's there just in case. However, we would advise you to at least open a Terminal and use the `ls` command to list a directory, and `cd` to change directory.

Linux isn't Windows

One thing to keep in mind when you first use Linux is that it's not Windows or Mac OS X. This largely means things you're used to in Windows won't work the same in Linux, or be in the same place, nor does it offer the same set of programs. So big commercial products such as Microsoft Office, Adobe Photoshop and development tools such as Microsoft Visual Studio aren't directly made for Linux – you can run them via a system called Wine – but the Linux open-source community has created its own tools such as LibreOffice, Gimp, Krita and a whole range of freely available and open-source products that offer the same capabilities.

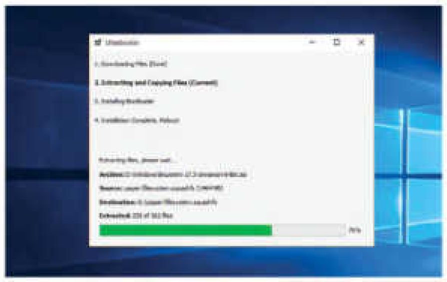


Mint on a USB drive



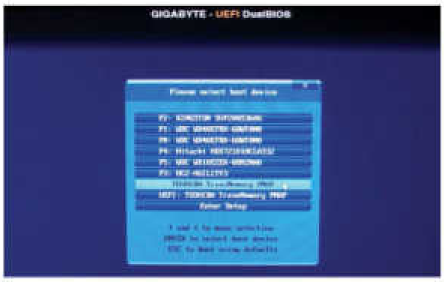
1 UNetbootin Linux

To run Mint from a USB stick, you first need a USB drive at least 4GB in size – 16GB would be ideal. You'll need the Mint ISO file from **www.linuxmint.com**, as discussed in the VirtualBox walkthrough, and we'll use the download tool UNetbootin from **http://unetbootin.github.io**. This installs the live disc ISO file directly to your USB drive.



2 Install Mint

The tool can directly download the Mint ISO image for you, but it's best practice to do this yourself, then you have a local copy to hand if you want to create a DVD copy or redo the process. Select Diskimage and locate the file in the Download folder. Ensure you have the correct USB drive selected in the pull-down menu and click OK to create the drive.



3 Boot and run

You can now boot your PC from the USB drive. However, you need to ensure your PC selects the USB drive as the boot device. Usually, when you first turn on your PC, a message says to press F11 or F12 to select the boot device. Some PCs have their own specific button – consult your manual or manufacturer. Linux Mint will now run.

Linux distro guide

A tantalising taste of the many different flavours of Linux.

In the Linux world, because it's free software that can be redistributed by absolutely anyone, this has lead to a proliferation of different versions of the Linux OS appearing. There are Linux distros tailored to a host of common uses, from hardcore Linux geeks to servers, science, media editing and so many more. The majority are general-use distros, which aim themselves at different types of users, such as experienced hands-on Terminal types, complete beginners, businesses, those with older PCs and people who want modern fancy interfaces. The choice can be bewildering, so to help you get started if you're interested in finding out more about Linux, here's our rundown of the major distro groups.

Debian

www.debian.org

One of the older distributions on the block, Debian is also technically the father of the most number of spin-off distros, including one of the most popular in the world – Ubuntu (see below). Debian itself can be thought of a bare-bones distro, because it comes with just enough to get it installed and up and running. It's really designed for servers and experts, but it's very stable and has complete repositories of software, which means that it is also very easy to extend.

Ubuntu

www.ubuntu.com

This is arguably the most popular – or at least the most widely known – Linux distro in the world. As we mentioned above, Ubuntu was spun out of the Debian project to create an easy-to-use distribution that would be suitable for anyone and everyone to use. Over the years, a huge number of spin-offs have been created – both official, such as Ubuntu Server, but also unofficial – because it became such an easy base to start from, with an easy installation, easy interface and easy software centre.

Mint

www.linuxmint.com

Based on Ubuntu, Mint took the crown of the most popular distro after Ubuntu moved to a more modern, touch-like desktop, whereas most Linux users wanted a traditional keyboard/mouse design, with a program menu and desktop icons. Mint offers the simplicity of installation, ease of use and the large software base of Ubuntu, but with optimised versions for older, slower computers, and a more fancy version that's known as Cinnamon.

Red Hat

www.redhat.com

It's highly unlikely you'll come across Red Hat as it's the big-business distro used by enterprises and corporations. It is, however, worth mentioning because it funds a couple of high-quality distros that can be freely used. Red Hat is a billion-dollar business and is, unusually, a paid-for Linux distro, but there are ways of effectively getting the same technology with a freely available distro.

Fedora

http://getfedora.com

The Fedora project was created by Red Hat as a test platform for cutting-edge Linux technology. Features are tested in Fedora and, when they're deemed stable enough, they are merged into the Red Hat distribution. This isn't as scary as it sounds, as Fedora only gets these features when they're stable. It's an excellent way of testing the latest technologies because Fedora tends to get them before most other distros.

OpenSUSE

www.opensuse.org

Another business-orientated distro that also has a long-standing pedigree. It might not be the sexiest distro on the block but it's widely used and often ranked in the top five distros, as it's super-stable and can be put to many uses, from standard desktop use to server. This is largely because it's another corporate-sponsored project and is much admired by developers and software vendors.

Mageia

www.mageia.org

Showing you how fluid the Linux distro world is, Mageia was created off the back of a long-running commercial Linux distro called Mandriva, which eventually went bust. From its ashes comes a fresh and powerful desktop distro, which offers a powerful but easy-to-use desktop and complete software library. It stands alongside Ubuntu and Mint for its ease of use yet manages to bring new features, too.

Arch

www.archlinux.org

A super-advanced version of Linux that you almost have to build from scratch. This means you need to be an expert to have any luck with it, but it also means you get an awesome OS that's exactly what you want. A big advantage is that Arch's active community of experts delivers the latest builds of software before anyone else gets them.

Manjaro

https://manjaro.github.io/

Arch Linux but without all the complexity. The Manjaro community has taken the base of Arch and created a pre-built Linux distro that is constantly updated, also called a rolling-release distro. While it does lean to the more technical side of distro releases, the developers have gone out of their way to try to make it accessible to all levels of users.

SteamOS

http://store.steampowered.com/steamos

Here's more of an example of how Linux is used in a fully commercial way. SteamOS is created by Valve Software, the company behind the PC digital distribution system called Steam. SteamOS is based on Debian and is a custom OS that drives a front for the Steam Big Picture mode, integrating controllers, home streaming and the store into one experience that you can enjoy from a box under your TV. ■

Subscribe to LINUX FORMAT

Get into Linux today!

Choose your LINUX FORMAT package

Print £63 For 12 months

Every issue comes with a 4GB DVD packed full of the hottest distros, apps, games and a lot more.

Digital £45 For 12 months

The cheapest way to get *Linux Format*. Instant access on your iPad, iPhone and Android device.

On iOS & Android!

Bundle £77 For 12 months

» Includes a DVD packed with the best new distros.

» Exclusive access to the *Linux Format* subscribers-only area – with 1,000s of DRM-free tutorials, features and reviews.

» Every new issue in print and on your iOS or Android device.

» Never miss an issue.

Get all the best in FOSS

Every issue packed with features, tutorials and a dedicated Pi section.

Subscribe online today...

myfavouritemagazines.co.uk/LINsubs

Prices and savings quoted are compared to buying full priced UK print and digital issues. You will receive 13 issues in a year. If you are dissatisfied in any way you can write to us at Future Publishing Ltd, 3 Queensbridge, The Lakes, Northampton, NN4 7BF, United Kingdom to cancel your subscription at any time and we will refund you for all un-mailed issues. Prices correct at point of print and subject to change. For full terms and conditions please visit: myfavm.ag/magterms. Offer ends 12/04/2016

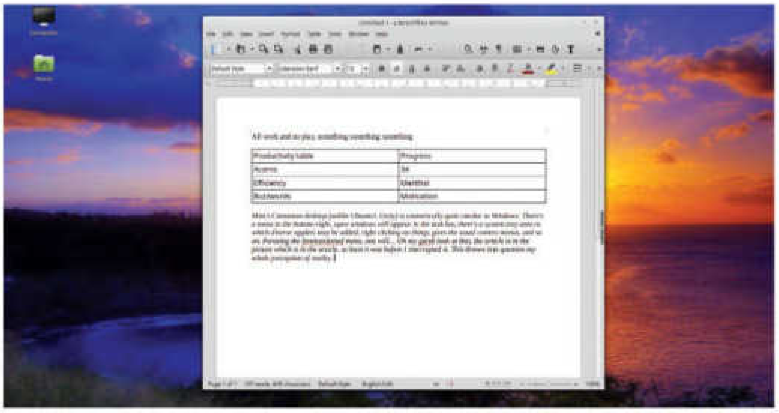
Get started with Mint and Python

Take a crash course in how to use the command line and follow our guide to create your very first Python program.

Mint's Cinnamon desktop (unlike Ubuntu's Unity) is cosmetically quite similar to Windows: there's a menu in the bottom-right; open windows will appear in the taskbar; there's a system tray area to which diverse applets may be added; right-clicking on things gives the usual context menus; and so on. Perusing the aforementioned menu, you will find a plethora of pre-installed software, including (but not limited to) the *LibreOffice* suite, the *Firefox* web browser, *VLC* for playing movies, *Gimp* for image manipulation and the *Transmission* BitTorrent client. Besides navigating the program groups in the Places menu, you can quickly access things by typing a few characters into the livesearch box.

The idea of working at the command line sends shivers down the spines of many people, not least because it stirs up long-repressed memories of darker times, viz. MS-DOS. But don't worry, it's a very powerful way to work, and thanks to tab-completion, you won't have to painstakingly transcribe lengthy commands or pathnames. Much of your day-to-day computing can be done at the terminal. File management, text editing and music playing can all be done with not one single click. In fact, pretty much anything you might be used to doing in a GUI can be done – or at least instigated (for example, playing a video) – from the command line.

> **"Pretty much anything you might be used to doing in a GUI can be done from the command line."**



❯ *LibreOffice Writer* is a more than adequate word processor for almost everybody's needs. It's compatible with *Microsoft Word* files.

We shall begin this tutorial by opening the Terminal program from the menu. We find ourselves confronted with an arcane prompt of the form

```
user@host:~$
```

and a rather menacing flashing cursor. The `~` is shorthand for our **home** directory, which lives at **/home/user/** in the Linux directory hierarchy. The `$` shows that we do not have root access, which limits the amount of damage we can do. Root is the superuser in Linux and, as such, access to the account is limited. On a standard Linux Mint installation, root access is granted through the `sudo` command (take a look at **https://xkcd.com/149/**), which temporarily elevates our privileges. If things were arranged differently, we would see that a root shell has not a `$` in its prompt, but a `#`. Let us ignore our hungry cursor no more, and start doing some command line-fu.

If we want to see a listing of all the files in the current directory, we use the `ls` command. So if we type that in and hit Enter, we'll see that we've got directories for Documents, Downloads, Music and a few others. We can access the directory one level up through the `..` operator. So if we do

```
$ ls ..
```

we will see a listing of the directory **/home/** in which we'll see our user's directory, together with the directories of any other users that are on the system (of which there probably none). If required, we can get more information by using the `-l` (long) option:

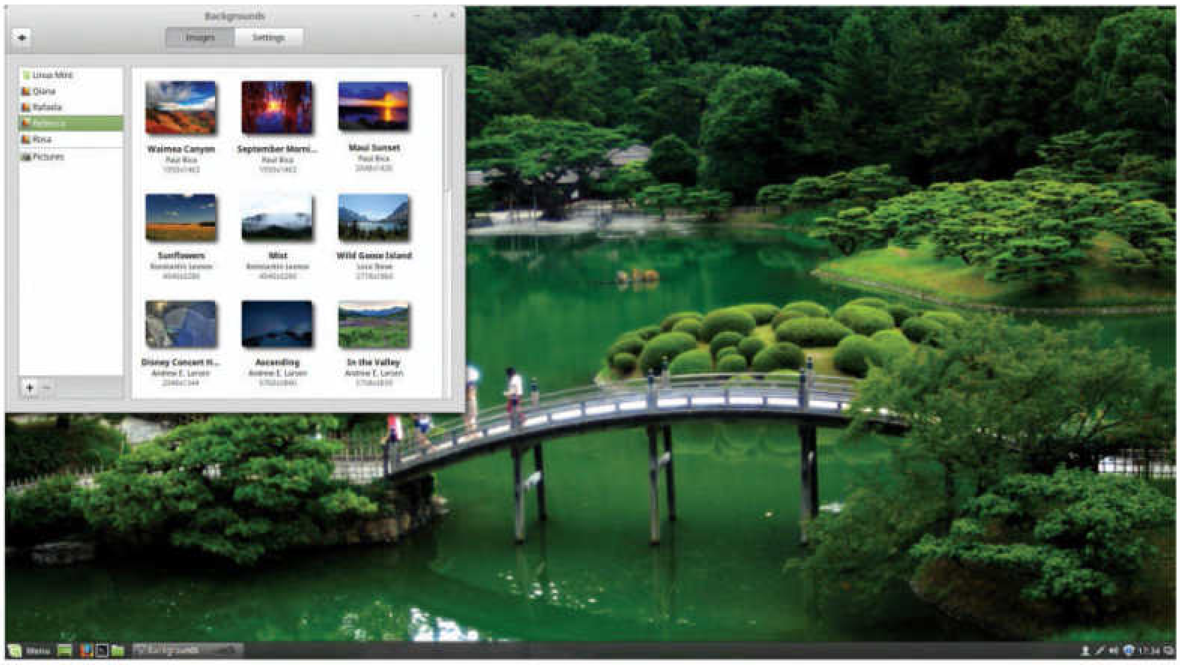
```
$ ls -l
```

This shows us file metadata such as permissions, ownership information, last modified date and file size.

We can navigate the directory hierarchy using the `cd` command. For example, to jump to the very top of the filesystem and see what the view is like from up there, you should issue:

```
$ cd /
$ ls
```

We see all kinds of tersely named directories, but we recognise **/home/** from being there a couple of sentences ago. The **/etc/** directory houses configuration files, which advanced users enjoy manipulating, but apart from that, there's generally no reason to do anything with the filesystem outside of our home directory – it's all taken care of by Mint's



› Mint comes bundled with some pretty wallpapers, which you peruse by right-clicking the desktop and choosing Change Desktop Background.

package manager. That said, it's fun to explore and familiarise yourself with how Linux does its filing. For example, the **/dev/** directory contains device nodes representing all the hardware attached to your system. Disk partitions get names such as **sda1** or **sdb2**, and the primary video card lives in **/dev/dri/card0**. This is quite a radical idea to digest, but on Linux, everything is a file.

For a more thorough look at the command line, see the five-minute introduction at **http://community.linuxmint.com/tutorial/view/100**. We're going to focus our attention towards Python programming now. We can return to our home directory at any time simply by calling `cd` with no arguments. So let's do that to start with, and then create a new directory in which to store our Pythonic labours.

```
$ cd ~
$ mkdir python
```

Mint currently uses the older Python 2.7 by default but it's worth adopting the newer 3.x series, unless you're morally opposed. So start the Python 3 interpreter by typing:

```
$ python3
```

We are presented with some version information, terse guidance and a different prompt: **>>>**. This enables us to enter Python commands and have the results immediately returned to us. It's known as a Read-Evaluate Print Loop (REPL). For example, we can use it as we would a standard calculator to perform basic arithmetic:

```
>>> 32 * 63
2016
>>> 9 ** 2
81
>>> 1 / 3
0.3333333333333333
```

We've put spaces around each operator, which is not necessary but does improve readability. And readable code is A Good Thing. But it's not just numbers that we can add up – we can do it with strings, too. We call them strings because they are strings of characters, but that doesn't stop them being added or multiplied:

```
>>> 'hello' * 3
'hellohellohello'
>>> 'hello there, ' + 'friend'
'hello there, friend'
```

Notice that we enclose strings in quotes. This is so they don't get confused with actual programming directives or variable names. Since strings can contain spaces, it's easy to see how such confusion may arise. There isn't a reasonable notion for subtracting or dividing by strings, so attempting to do that results in an error. Programmers encounter diverse error messages as a matter of course, so you may as well try that now. Your first ever type error – congratulations! Exit the interpreter by pressing Ctrl+D or typing `exit()`. The interpreter is good for trying out code snippets but it's no use for proper coding. We're going to create our first Python program and run it, all from the command line.

All programming begins with simple text files, and we can use the humble *nano* editor to create one. Let's navigate to our newly created directory and create our Python script:

```
$ cd ~/python
$ nano helloworld.py
```

Python programs all use the .py extension, and our particular program, as is tradition in such matters, is going to issue a worldly greeting. The *nano* text editor is pretty easy to work with – you can press Ctrl+X at any time to exit (you'll be prompted to save any changes). There are some helpful shortcuts displayed on the bottom for searching (Ctrl+W), saving (Ctrl+O) and the like. If you've never used a terminal-based text editor, then you might dismayed to learn that the mouse can't be used as you might expect here. It is possible to select text and paste it using the middle mouse button, but positioning of the cursor is done with – and only with – the cursor keys.

Getting Python to print messages to the terminal is easy – we use the `print()` function. Type the following into *nano:*

```
print('Hello world')
```

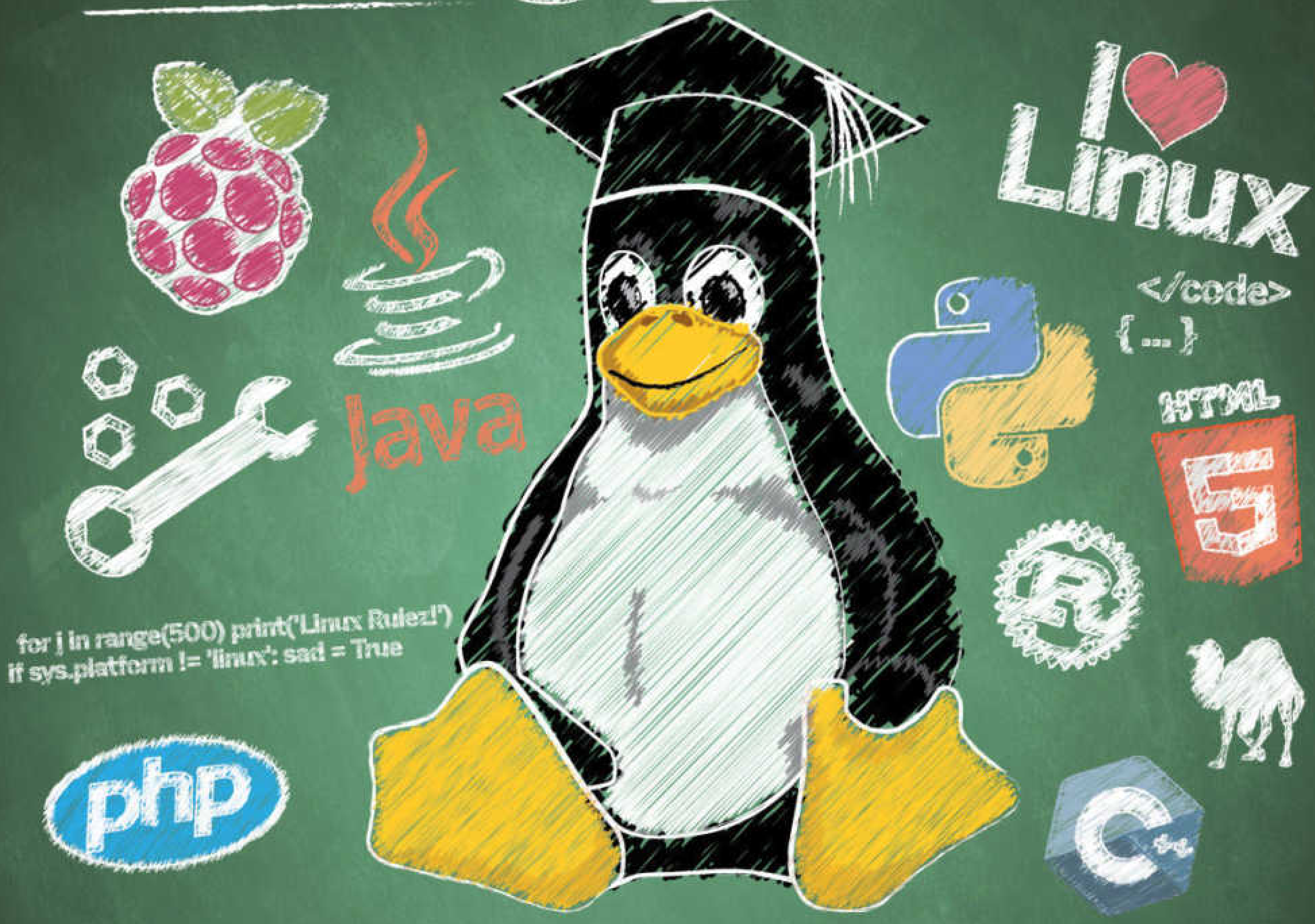
As before, we've had to enclose our string in quotes. Now commence departure from *nano* with Ctrl+X, press Y to save and then press Return to use our existing file. Back at the command line, the moment of truth – let's see whether Python can run our program:

```
$ python3 helloworld.py
```

If all goes to plan, a friendly greeting should be displayed. If that didn't happen, welcome to the world of programming, in which careless typing is punished. Rinse, lather and repeat the editing and running stages until it works. Then settle down with a cup of tea, satisfied in the knowledge that you wrote a program and lived to tell the tale. ■

FREE DVD!
MINT 17.3: TRY IT NOW!
LINUX FORMAT
Get into Linux today!
65 Pages of tutorials and features
» Defend servers with Kali
» Encode your HD video
» Master VirtualBox 5
Coding Academy: WebGL explored, MongoDB primer
ESCAPE WINDOWS
Discover the freedom and power of Linux
» Free to install » Easy to use
» Fast, secure and powerful
» Get started right now!
OUT NOW!
MINT 17.3
NEW RELEASE
The perfect distro for new users – works on any PC!
» Fully updated » Cinnamon and Mate » Improved interface
32- & 64-bit
OpenSUSE Leap
NEW RELEASE
Step up to the rock solid enterprise-class distro
» Kernel 4.1 » KDE desktop
» Snapshots » New Btrfs
64-bit
LIVE DISC: READY TO RUN
EVERYTHING YOU NEED TO GET STARTED WITH LINUX
Plus: Pi User
8-page Raspberry Pi companion
» Games and GPIO in Scratch
» Discover OpenMediaVault
Future

Join Tux's Coding Academy


I ♥ Linux
</code>
{...}
HTML
5
Java
for j in range(500) print('Linux Rulez!')
if sys.platform != 'linux': sad = True
php
C++

Coding isn't scary. Promise. If you've always wanted to learn, jump straight in with our quick-start guide.

Coding is the new cool. If you don't know how to Python, all your friends are going to be laughing at you…

We're not being flippant. Knowing how to code is almost an essential for the open-source fiend. Be it basic Bash scripting or knowing how to read a bit of PHP, coding knowledge is more than useful – it's an essential skill, particularly now that coding is being taught in schools.

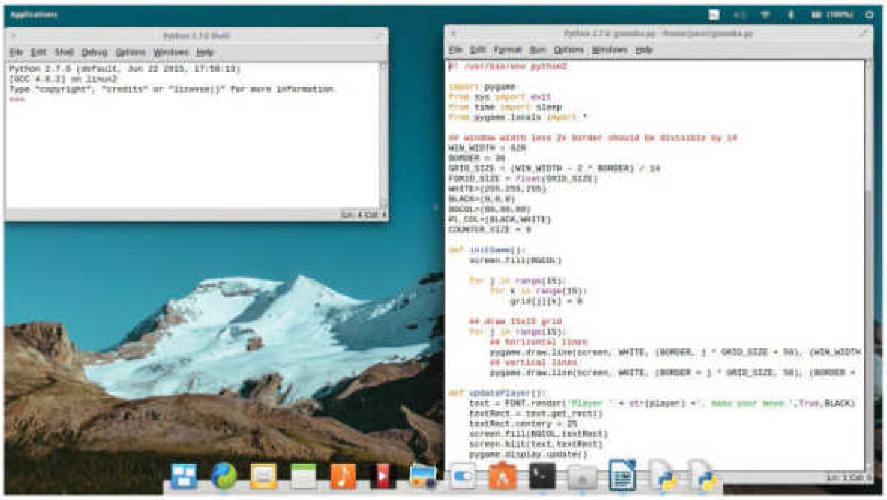
Knowing a bit of code helps with using Linux itself, helps you get more out of the terminal, can open doors to a new career or help you troubleshoot why that webpage won't work correctly. Other than taking a bit of time to pick up, there's also no cost and, as a FLOSS user, access to every language under the sun is just waiting an **apt-get** away.

So grab our hand and let's take just a few minutes to create a fun game in Python, learn which languages are right for you and your projects, then see how you can tackle the new school curriculum and web development.

»

Get with the program

This is going to be a very gentle introduction to programming in Python, in which we'll explore the basics of the language, then use the *Pygame* module to make a simple game. That we are able to do this is testament to the power of Python and *Pygame* – much of the tedium inherent in game work is abstracted away and, once we've covered some elementary programming constructions, for the most part we work with intuitive commands.



```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>>
```

```
#! /usr/bin/env python2

import pygame
from sys import exit
from time import sleep
from pygame.locals import *

## window width less 2x border should be divisible by 14
WIN_WIDTH = 620
BORDER = 30
GRID_SIZE = (WIN_WIDTH - 2 * BORDER) / 14
FGRID_SIZE = float(GRID_SIZE)
WHITE=(255,255,255)
BLACK=(0,0,0)
BGCOL=(80,80,80)
P1_COL=(BLACK,WHITE)
COUNTER_SIZE = 8

def initGame():
    screen.fill(BGCOL)

    for j in range(15):
        for k in range(15):
            grid[j][k] = 0

    ## draw 15x15 grid
    for j in range(15):
        ## horizontal lines
        pygame.draw.line(screen, WHITE, (BORDER, j * GRID_SIZE + 50), (WIN_WIDTH
        ## vertical lines
        pygame.draw.line(screen, WHITE, (BORDER + j * GRID_SIZE, 50), (BORDER +

def updatePlayer():
    text = FONT.render('Player '+ str(player) +', make your move.',True,BLACK)
    textRect = text.get_rect()
    textRect.centery = 25
    screen.fill(BGCOL,textRect)
    screen.blit(text,textRect)
    pygame.display.update()
```

Applications
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Ln: 4 Col: 4
File Edit Format Run Options Windows Help
Ln: 1 Col: 0

The *IDLE* development environment is purpose-built for Python. You can install it on Ubuntu (or in this case, ElementaryOS) with apt-get install idle .

Follow the instructions in the box below to check which version of Python you have and install *Pygame*. Whether on a Raspberry Pi or a larger machine, start at the command line. So open up a terminal (*LXTerminal* in Raspbian) and start Python 2.7 with:

```
$ python2
```

Alternatively, start up *IDLE* and start a command prompt from there by choosing Python 2.7 Shell from the Window menu. Both methods start the Python interactive interpreter, wherein Python commands can be typed at the `>>>` prompt and evaluated immediately. It is here that we will forge our first Python program, by carefully typing out the following incantation:

```
>>> print('Hello World')
```

As you might suspect, pressing Enter causes Python to display a global greeting. You can exit the interpreter at any point by pressing Ctrl+D or using the `exit()` command. For larger programs, it makes sense to work in a text editor, or an Integrated Development Environment (like *IDLE),* but the interpreter is a great way to test smaller code fragments. So we'll use it to now to introduce some fundamental coding concepts and constructs. First, let's introduce the idea of variables:

```
>>> name = 'Methuselah'
>>> age = 930
>>> print(name, ' is ', age, ' years old.')
```

We can assign values to variables, change them to our hearts' content, and use `print` statements to see them. Technically, the brackets in the `print` line are only required in Python 3, but they don't do any harm and it's good practice to write, wherever possible, ambidextrous code that will run in both versions. Variables in Python are automatically assigned a type based on their content. So `name` is a string (short for a string of characters) and `age` is an integer (a whole number). You can check this by typing `type(name)` and so on.

Some types can be coerced to other types – we can transmute `age` to a floating point number (one with a decimal part) with:

```
>>> age = float(age)
>>> age
```

Just typing the variable name into the interpreter will show you its value, so you can easily see the changes you have enacted. We can convert ints or floats to strings, using the function `str()` . Python can also go the other way, converting a string to a float, for example, but this will only work if the original string looks something like its target type: for example, `float('10.0')` will work, but `float('Rumplestiltskin')` will not.

Just as division works differently for floats and ints, so addition works differently for strings. Here the `+` operator stands for concatenation, tacking the latter string on to the end of the former. Thus:

```
>>> 'Hello ' + 'world'
'Hello world'
>>> str(123) + str(456)
'123456'
>>> 'Betelgeuse' * 3
'BetelgeuseBetelgeuseBetelgeuse'
```

The last line shows that we can also multiply strings – division and substraction, however, are not defined for strings.

Data types dictate how data is represented internally, and the effects of this can be quite subtle. For example, in Python 2 the division operator `/` works differently if one of its arguments is a float:

Installing Python and Pygame

If you're using Raspbian on the Raspberry Pi or any flavour of desktop Linux, then the chances are that you already have at least one (probably two) versions of Python installed. While the latest release (1.9.2) is now Python 3-compatible, no distributions are shipping this version yet, so we'll stick with Python 2 (2.7 to be precise) for this tutorial. Check your default Python version by typing:

```
$ python -V
```

If this returns a result that begins with 2, then everything is fine. If, on the other hand, you see 3-point-something, then check Python 2 availability with the command:

```
$ python2 -V
```

Some distros, notably Arch Linux and the recently released Fedora 22, don't ship the 2.7 series by default. Installing *Pygame*, however, will pull it in as a dependency, so let's do that now. Users of Debian-derived distributions (including Raspbian on the Pi) should use the following command to install *Pygame*:

```
$ sudo apt-get install python-pygame
```

Users of other distributions will find a similarly named package (on Arch it's called **python2-pygame**) and should be able to install it through the appropriate package manager – *pacman yum, zypper* or whatever. Most distributions bundle the *IDLE* environment with each version of Python installed; if you can't find an icon for it, try running the commands `idle` or `idle2` . If that fails to produce the goods, then go hunting in your distro's repos.

```
>>> 3/2
>>> 1
>>> 3/2.
>>> 1.5
```

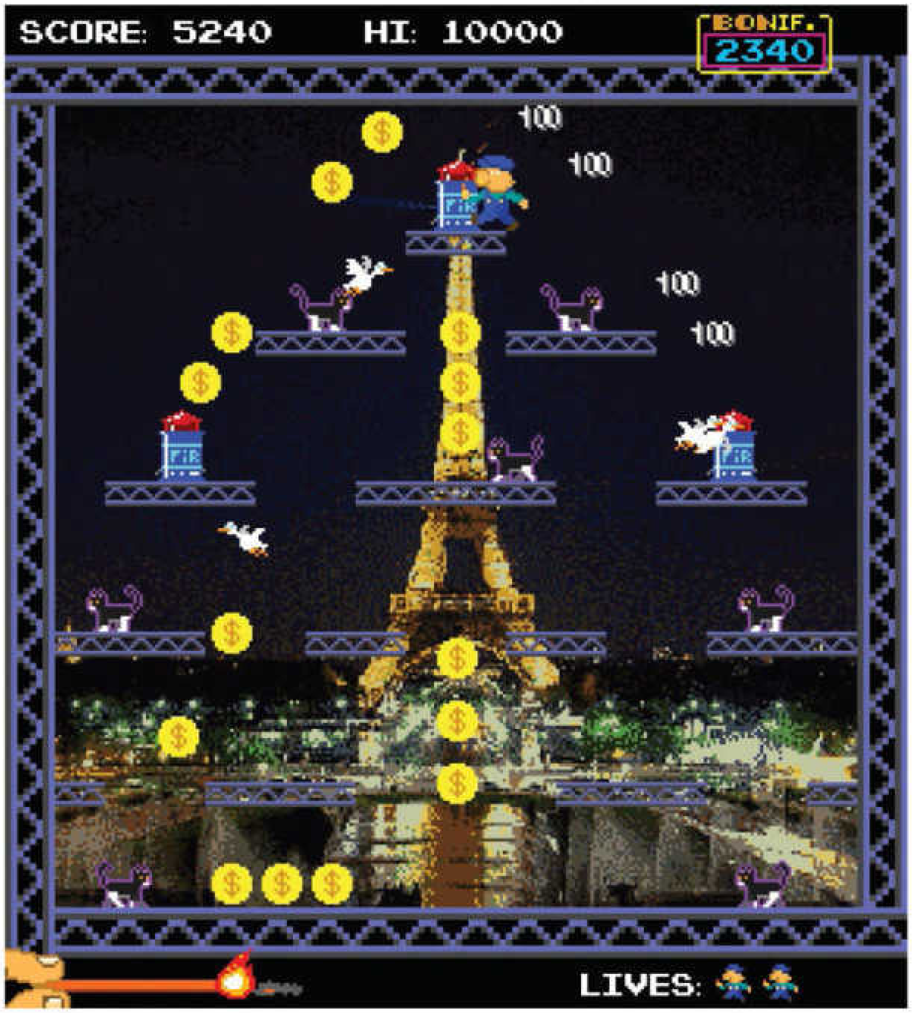
Funny the difference a dot can make. Note that we have been lazy here in typing simply `2.` when we mean `2.0`. Python is all about brevity. (Besides, why make life hard for yourself?) Sooner or later, you'll run into rounding errors if you do enough calculations with floats. Check out the following doozy:

```
>>> 0.2 * 3
0.6000000000000001
```

Such quirks arise when fractions have a non-terminating binary decimal expansion. Sometimes, these are of no consequence, but it's worth being aware of them. They can be worked around either by coercing floating point variables to ints, or using the `round()` function, which will give you only the number of decimal places you require. We'll see this in practice when we program our Gomoku game later.

Going loopy

Often, programmers desire to do almost the same thing many times over. It could be appending entries to a list, adding up totals for each row in a table, or even subtracting energy from all enemies just smitten by a laser strike. Iterating over each list item, table row or enemy manually would be repetitive and make for lengthy, hard-to-read code. For this reason, we have loops, like the humble `for` loop below. When you hit Enter after the first line, the prompt changes to `…` because the interpreter knows that a discrete codeblock is coming and the line(s) following the `for` construct 'belong' to it. Such codeblocks need to be indented, usually using four spaces, though you can use as many as you like so long as you're consistent. If you don't indent the second line, Python shouts at you. Entering a blank line after the `print` statement ends the codeblock and causes our loop to run.



❯ **There are all kinds of *Pygame*-powered games. This one, *You Only Get One Match*, features lots of fireworks but limited means of ignition. Check it out at http://bit.ly/LXF202-onematch.**

```
>>> for count in range(5):
...       print('iteration #', count)
```

There's a few things going on here. We have introduced a new variable, an integer by the name of `count`. Giving sensible names to your variables is a good idea; in this case, it implies (even in the absence of any other coding knowledge) that some counting is about to happen. The `range()` function, when used in isolation, returns a list consisting of a range of integers. We'll cover lists in just a moment, but for now we just need to know that `range(5)` looks like [0, 1, 2, 3, 4], which you can verify in the interpreter. So our variable `count` is going to iterate over each of these values, with the `print()` line being issued five times – once for each value in the range. Another type of loop is the `while` loop. Rather than iterating over a

»

How to play Gomoku

Gomoku is short for gomokunarabe, which is roughly Japanese for 'five pieces lined up'. The game, in fact, originated in China some 4,000 years ago. Players take turns to each place a counter on the intersections of a square grid with the goal of forming unbroken lines (horizontal, vertical or diagonal) of length 5. Traditionally, the board has 19x19 intersections, but we've gone for the smaller 15x15 board used in some variations. We haven't included an AI (that would be somewhat too complicated for a beginner tutorial), so you'll need to find a friend/other personality with which to play. Alternatively, there are plenty of online versions you can play, and KDE users get the similar *Bovu* game with their desktop.

It's easy to figure out some basic strategies, such as always blocking one side of your opponent's 'open three' line or obstructing a 'broken four'. Yet to become a master takes years of practice. The basic rule set as we've implemented it heavily favours the starting player (traditionally black). Work by L Victor Allis has shown that a good player (actually a perfect player) can force a win if they start. To mitigate against this, big tournaments use a starting strategy called swap2. The first player places two black counters and one white one on the board, and the second player either chooses a colour or places another black and another white counter on the board and allows player one to choose colours. You are free to modify the code to force use of swap2, but it's entirely possible to obey this rule without any code modification: just disobey the first few 'Player x, make your move' prompts.

list, our while loop keeps going over its code block until a condition ceases to hold. In the following example, the condition is that the user claims to have been born after 1900 and before 2016.

```
>>> year = 0
>>> while year < 1900 or year >= 2015:
...       year = input("Enter your year of birth: ")
...       year = int(year)
```

Again the loop is indented, and again you'll need to input a blank line to set it running. We've used the less than (<) and greater than or equal to (>=) operators to compare values. Conditions can be combined with the logical operators and , or and not . So long as year has an unsuitable value, we keep asking. It is initialised to 0, which is certainly less than 1900, so we are guaranteed to enter the loop. We've used the input() function, which returns whatever string the user provides. This we store in the variable year , which we convert to an integer so that the comparisons in the while line do not fail. It always pays to be as prudent as possible as far as user input is concerned: a malicious user could craft some weird input that causes breakage, which while not a big deal here, is bad news if it's done, say, on a web application that talks to a sensitive database. You could change 1900 if you feel anyone older than 115 might use your program. Likewise, change 2015 if you want to keep out (honest) youngsters.

The opposite of listless

We mentioned lists earlier, and in the exciting project that follows we'll use them extensively, so it would be remiss not to say what they are. Lists in Python are flexible constructions that store a series of indexed items. There are no restrictions on said items – they can be strings, ints, other lists, or any combination of these. Lists are defined by enclosing a comma-separated list of the desired entries in square brackets. For example:

```
>>> myList = ['Apple', 'Banana', 'Chinese Gooseberry']
```

The only gotcha here is that lists are zero-indexed, so we'd access the first item of our list with myList[0] . If you think too much like a human, then 1-indexed lists would make more sense. Python doesn't respect this, not even a little, so if you too think like a meatbag, be prepared for some classic off-by-one errors. We can modify the last item in the list:

```
>>> myList[2] = 'Cthulhu'
```

Lists can be declared less literally – for example, if we wanted to initialise a list with 100 zeroes, we could do:

```
>>> zeroList = [0 for j in range(100)]
```

This is what is known as a list comprehension. Another example is

```
>>> countList = [j for j in range(100)]
```

which results in a list containing the integers 0 up to and including 99, which could equally well be achieved with range(100) in Python 2. However, the concept is more powerful – for example, we can get a list of squares using the exponentiation (to the power of) operator ** :

```
>>> squareList = [j ** 2 for j in range(100)]
```

And after that crash course, we're ready to program our own game. You'll find all the code at **http://pastebin.com/FRe7748B**, so it would be silly to reproduce that here. Instead, we're focusing on the interesting parts, in some cases providing an easier-to-digest fragment that you can play with and hopefully see how it evolves into the version in the program. To dive in and see the game in action, download **gomoku.py** and copy it to your home directory, and run it with:

```
$ python2 gomoku.py
```

On the other hand, if you want to see some code, open up that file in *IDLE* or your favourite text editor. Starting at the first line is a reasonable idea... It looks like:

```
#!/usr/bin/env python2
```

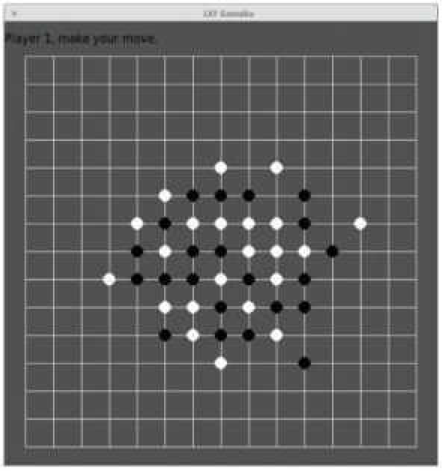
This line is actually ignored by Python (as are all lines that begin with #) but it is used by the shell to determine how the file should be executed. In this case we use the *env* utility, which should be present on all platforms, to find and arm the Python 2 executable. For this nifty trick to work, you'll need to make the **gomoku.py** file executable, which is achieved from the command prompt (assuming you've copied the file to your home directory, or anywhere you have write permission) with:

```
$ chmod +x gomoku.py
```

You'll find you can now start the game with a more succinct:

```
$ ./gomoku.py
```

Next we have three import statements, two of which (pygame and sys) are straightforward. The pygame module makes easy work of doing game-related things – we're really only scratching the surface with some basic graphics and font rendering. We need a single function, exit() , from the sys module so that we can cleanly shut down the game when we're done. Rather than importing the whole sys module, we import only this function. The final import line is just for convenience – we have already imported pygame, which gives us access to pygame.locals , a bunch of constants and variables. We use only those relating to mouse, keyboard and quitting events. Having this line here means we can access, for example, any mouse button events with MOUSEBUTTONDOWN without prefixing it with pygame.locals .



Player 1, make your move.

› **Many tense counter-based battles can be had with your very own self-programmed version of Gomoku.**

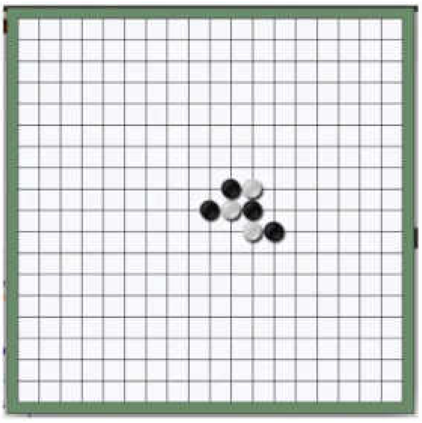
It's all in the (Py)game

Throughout the program, you'll notice that some variables are uppercase and some are not. Those in uppercase are either from pygame.locals or should be considered constants, things that do not change value over the course of the game. Most of these are declared after the import statements and govern things like the size of the window and counters. If you want to change the counter colours, to red and blue, for example, you could replace the values of WHITE and BLACK with (255,0,0) and (0,0,255) respectively. These variables are tuples (a similar structure to a list, only it can't be changed), which dictate the red, green and blue components of colours.

Next you'll see a series of blocks beginning with def: – these are function definitions and, as is the case with other codeblocks in Python, they are demarcated by indentation. The initGame() function initialises the play area. Here's a simple version that shows what this function does:

```
WIN_WIDTH = 620
GRID_SIZE = (WIN_WIDTH) / 14
WHITE=(255,255,255)
BLACK=(0,0,0)
BGCOL=(80,80,80)
def initGame():
  screen.fill(BGCOL)
  for j in range(15):
    pygame.draw.line(screen, WHITE, (0, j * GRID_SIZE), (WIN_WIDTH, j * GRID_SIZE))
    pygame.draw.line(screen, WHITE, (j * GRID_SIZE, 0), (j * GRID_SIZE, WIN_WIDTH))
pygame.init()
pygame.display.set_caption('LXF Gomoku')
screen = pygame.display.set_mode((WIN_WIDTH,WIN_WIDTH))
initGame()
pygame.display.update()
```

If you add the three import lines to the beginning of this, it is actually a perfectly



› **Joel Murielle's graphical *Gomoku* is available from the *Pygame* website. *Pygame* takes all the pain out of working with sprites – notorious troublemakers.**

valid Python program. The `initGame()` function doesn't do anything until it is called at the last line, by which time we've already initialised *Pygame*, set our window title, and set the window size to 620 pixels. All variables set up outside of function definitions – the five all-caps constants at the beginning and `screen` – are accessible inside function definitions; they are known as global variables. Variables defined inside function definitions are called 'local' – they cease to exist when the function exits, even if they have the same name as a global variable – again, something to be aware of in your future coding endeavours. The variable `screen` refers to the 'canvas' on which our game will be drawn, so it will be used extensively later on. The `initGame()` function's first act is to paint this canvas a delightful shade of grey (which you're very welcome to change). Then we use a loop to draw horizontal and then vertical lines, making our 15x15 grid. None of this artwork will appear until we tell *Pygame* to update the display, hence the last line.

Astute readers will notice that the grid overruns ever so slightly at the edges. This is because drawing 15 equispaced parallel lines divides the board into 14, but 620 (our window size) is not divisible by 14. However, when we add in some window borders – since we want to place counters on the edge lines as well – 620 turns out to be a very good number, and we were too lazy to change it. Although rough around the edges, it's still testament to *Pygame's* power and Python's simplicity that we can do all this in just a few lines of code. Still, let's not get ahead of ourselves – our game still doesn't do anything.

Finer points

From here onwards, we'll refer to the actual code, so any snippets we quote won't work in isolation – they're just there to highlight things. You'll notice that the `FONT` variable isn't defined with the other constants; this is because we can't use *Pygame's* font support until after the *Pygame's* `init()` method has been called. Let's look at the main game loop right at the end of the code. The introductory clause `while True:` suggests that this loop will go on for ever. This is largely correct – we want to keep checking for events, namely mouse clicks or the user clicking the exit button, until the game is done. Obviously, we exit the loop when the application quits – clicking the button triggers a `QUIT` event, which we react to with the `exit()` function from the sys package. Inside the main loop, the first thing we do is call the `updatePlayer()` function, which you'll find on line 32. This updates the text at the top of the screen that says whose go it is, drawing ('blitting') first a solid rectangle so any previous text is erased.

Next we loop over the events in the events queue; when the player tries to make a move, the `tryCounterPlace()` function is called, with the mouse coordinates passed along. To keep track of the game, we use a two-dimensional square array (a list of lists) in the variable `grid`. This is initialised as all 0s, and when a player makes a move, a 1 or a 2 is entered accordingly. The first job of the `tryCounterPlace()` function is to reconcile the mouse coordinates where the user clicked with a pair of coordinates with which to index the grid variable. Of course, the user may not click exactly on an intersection, so we need to do some cheeky rounding here. If the player clicks outside of the grid (e.g. if they click too far above the grid, so the `y` coordinate will be negative) then the function returns to the main loop. Otherwise, we check that the grid position is unoccupied and if so draw a circle there, and update our state array `grid`. A successful move causes our function to return a `True` value, so looking at line 111 in the code, we see this causes the next player's turn. But before that is enacted, by the `updatePlayer()` call at the top of the loop, we call the `checkLines()` function to see if the latest move completed a winning line. Details of how this check is carried out are in the box.

When a winning counter is detected by our state-of-the-art detection algorithm, the `winner()` function is invoked. This replaces the text at the top of the screen with a message announcing the victor, and the gameover loop is triggered. This waits for a player to push R to restart or rage quit. If a restart is ordered, the player order is preserved and, as this is updated immediately before `checkLines()` is called, the result is that the loser gets to start the next round.

This is a small project (only about 120 lines, not a match for the 487-byte *Bootchess* you can read about at **www.bbc.co.uk/news/technology-31028787**), but could be extended in many ways. Graphics could be added, likewise a network play mode and, perhaps most ambitiously, some rudimentary AI could be employed to make a single-player mode. This latter has already been done…

»

Reading between the lines

Part of Key Stage 2 involves learning to understand and program simple algorithms. We've already covered our basic game flow algorithm – wait for a mouse click (or for the user to quit), check whether that's a valid move, check if there's a line of five, and so on. At the heart of that last stage lies a naïve, but nonetheless relevant, algorithm for detecting whether a move is a winning one.

Consider the simpler case where we're interested only in horizontal lines. Then we would loop over first the rows and then the columns of our grid array. For each element, we would check to see that it is non-zero (i.e. there is a counter there) and whether the four elements to its right have the same value. In Python, it would look like this:

```
for j in range(15):
  for k in range(10):
    pl = grid[j][k]
    if pl > 0:
      idx = k
      while grid[j][idx] == pl and idx < 14:
        idx += 1
      if idx - k >= 5:
        # game winning stuff goes here
```

Note that the inner loop variable `k` reaches a maximum value of only 9. We do not need to check row positions further right than this because our algorithm will reach out to those positions if a potential line exists there. Our variable `idx` effectively measures the length of any line; it is incremented using the `+=` operator, short for `idx = idx + 1`.

The algorithm is easily adapted to cover vertical and diagonal lines. Rather than four separate functions, though, we've been clever and made a general function `lineCheck()`, which we call four times with the parameters necessary for each type of line checking. Said parameters just change the limits of the `for` loops and how to increment or decrement grid positions for each line direction.

Languages: An overview

One of technology's greatest achievements was IBM's Fortran compiler back in the 1950s. It allowed computers to be programmed using something a bit less awkward than machine code. Fortran is still widely used today and, while some scoff at this dinosaur, it remains highly relevant, particularly for scientific computing. That said, nobody is going to start learning it out of choice, and there are all manner of other languages out there.

Traditionally, you have had the choice between hard and fast languages – such as Java, C and C++ – or easy and slower ones, such as Python or PHP. The fast languages tend to be the compiled ones, where the code has to be compiled to machine code before it can be run. Dynamic languages are converted to machine code on the fly. However, on some level, all programming languages are the same – there are some basic constructs such as loops, conditionals and functions, and what makes a programming language is simply how it dresses these up.

For those just starting coding, it's simply baffling. Opinions are polarised on what is the best language to learn first of all, but the truth is that there isn't one, though for very small people we heartily recommend Scratch. Any language you try will by turns impress and infuriate you. That said, we probably wouldn't recommend C or Haskell for beginners.

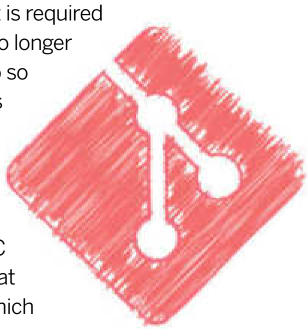
> **"Although any language will by turns impress and infuriate you, Python has a lot going for it."**

There is a lot of popular opinion that favours Python, which we happily endorse, but many are put off by the Python 2 versus 3 fragmentation. Python has a lot going for it: it's probably one of the most human-readable languages out there. For example, you should, for readability purposes, use indentation in your code, but in Python it's mandatory. By forcing this issue, Python can do away with the curly brackets used by so many other languages for containment purposes. Likewise, there's no need to put semicolons at the end of every line. Python has a huge number of extra modules available, too – we've seen *Pygame,* and our favourite is the API for programming *Minecraft* on the Pi.

Beginner-friendly

Other languages suitable for beginners are JavaScript and PHP. The popularity of these comes largely from their use on the web. JavaScript works client-side (all the work is done by the web browser), whereas PHP is server-side. So if you're interested in programming for the web, either of these languages will serve you well. You'll also want to learn some basic HTML and probably CSS, too, so you can make your program's output look nice, but this is surprisingly easy to pick up as you go along. PHP is cosmetically a little messier than Python, but soon (as in *The Matrix)* you'll see right through the brackets and dollar signs. It's also worth mentioning Ruby in the accessible languages category. It was born of creator Yukihiro Matsumot's desire to have something as "powerful as Perl, and more object-oriented" than Python.

Barely a day goes by without hearing about some sort of buffer overflow or use-after-free issue with some popular piece of software. Just have a look at **https://exploit-db.com**. All of these boil down to coding errors, but some are easier to spot than others. One of the problems with the fast languages is that they are not memory safe. The programmer is required to allocate memory as it is required and free it when it's no longer required. Failure to do so means that programs can be coerced into doing things they shouldn't. Unfortunately, 40 years of widespread C usage have told us that this is not a task at which humans excel, nor do we seem to be getting any better at it. Informed by our poor record, a new generation of languages is emerging. We have seen languages from Google (Go), Apple (Swift) and Mozilla (Rust). These languages all aim to be comparable in speed to C, but at the same time guaranteeing the memory safety so needed in this world rife with malicious actors.



Rust recently celebrated its 1.0 release, and maybe one day *Firefox* will be written using it, but for now there are a number of quirks and pitfalls that users of traditional languages are likely to find jarring. For one thing, a program that is ultimately fine may simply refuse to compile. Rust's compiler aims for consistency rather than completeness – everything it can compile is largely guaranteed, but it won't compile things where any shadow of doubt exists, even when that shadow is in the computer's imagination. So coders will have to jump through some hoops, but the rewards are there – besides memory safety and type inference, Rust also excels at concurrency (multiple threads and processes), guaranteeing thread safety and freedom from race conditions.

Programming paradigms and parlance

With imperative programming, the order of execution is largely fixed, so that everything happens sequentially. Our Gomoku example is done largely imperative style, but our use of functions makes it more procedural – execution will jump between functions, but there is still a consistent flow.

The object-oriented (OO) approach extends this even further. OO programs define classes, which can be instantiated many times; each class is a template for an object, which can have its own variables (attributes) and its own code (methods). This makes some things easier, particularly sharing data without resorting to lengthy function calls or messy global variables. It also effects a performance toll, though, and is quite tricky to get your head around. Very few languages are purely OO, although Ruby and Scala are exceptions. C++ and Java support some procedural elements, but these are in the minority. Functional programming (FP) has its roots in logic, and is not for the faint-hearted. This very abstract style is one in which programs emphasise more what they want to do than how they want to do it. Functional programming is all about being free of side-effects – functions return only new values, there are no global variables. This makes for more consistent languages such as Lisp, Scheme, Haskell and Clojure. For a long time, functional programming was the preserve of academia, but it's now popular within industry.

Coding in the classroom

In September 2014, the UK embarked on a trailblazing effort that saw coding instilled in the National Curriculum. When the initiative was announced in 2013, then education secretary Michael Gove acknowledged that the current ICT curriculum was obsolete – "about as much use as teaching children to send a telex or travel in a zeppelin". Far more important was imparting coding wisdom unto the young padawans. Coding skills are much sought after, as evidenced by industry consistently reporting difficulties in finding suitably qualified applicants for tech jobs in the UK.

The 'Year of Code' was launched to much fanfare, though this was slightly quelled as details emerged: a mere pittance was to be added to the existing ICT allocation, and most of this would be spent on training just 400 'Master Teachers' who could then pass on their Master Skills to lesser teachers around the country. Fans of schadenfreude will enjoy the BBC *Newsnight* interview with the then Year of Code chief, wherein she admits not knowing how to code, despite claiming it is vital for understanding how the world works.

› The FUZE box gives you everything you need to start fiddling with registers or making GPIO-based mischief.

Learning opportunities

Criticism and mockery aside, we're genuinely thrilled that children as young as five are, even as we speak, learning the dark arts of syntax, semantics and symbolism. Fear now, ye parents, when your progeny hollers at you (that's what kids do, apparently), "What's the plural of mongoose?" and then follows through seeking clarification on the finer points of recursion and abstraction. Rightly or wrongly, many private firms will benefit from the concerned parents and confused kids resulting from the new computing curriculum. But there are more wholesome directions in which one can seek help.

> **"Without any coding, the ICT curriculum was 'as much use as teaching kids to send a telex'."**

For one thing, you'll always find great tutorials monthly in magazines such as *Linux Format* – **https://linuxformat.com**. There are also many free resources on the web. Some of them (such as the official Python documentation) are a little dry for kids, but we'd encourage adults to learn these skills alongside their offspring. Refurbishing an old machine with a clean Linux install will provide a great platform to do just this. All the software you need is free, and distributions such as Mint and Ubuntu are easy enough to get to grips with.

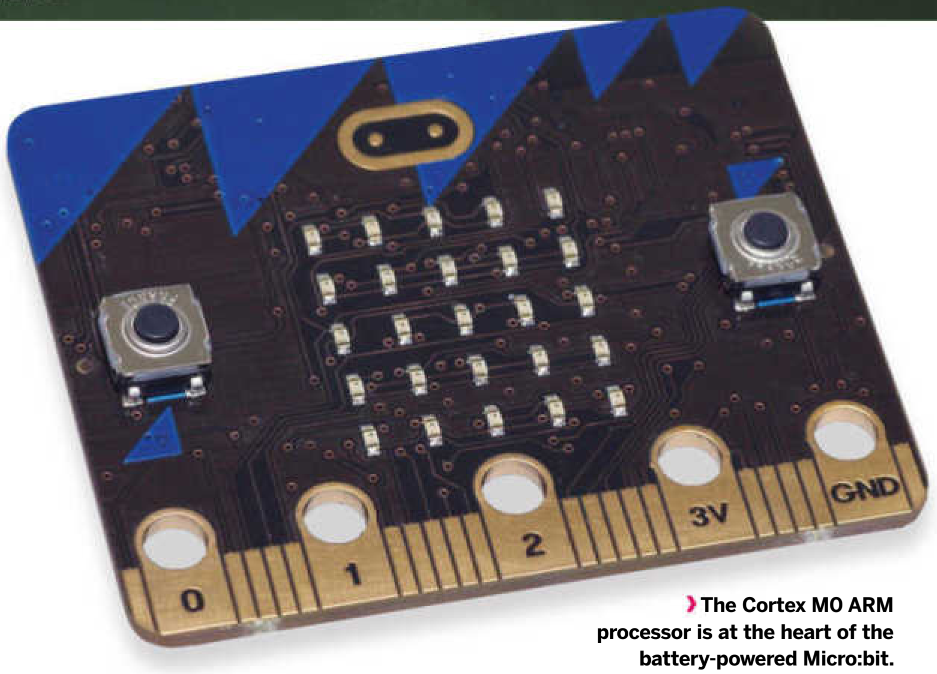
Besides a stray PC, the Raspberry Pi (*see page 82 for more*) is another great way to provide a learning platform. If you're willing to settle for the older B+ model, then you can get one for about £25, and it'll plug straight into your telly. Of course, you'll need to scavenge a keyboard, mouse, SD card, HDMI cable and possibly a wireless adapter, too, if trailing an Ethernet cable to your router is not an option. Mercifully, there are many kits available (for example, the Kano, billed as a DIY computer), which will provide these additional peripherals, not to mention a glorious array of Pi cases available to protect it from dust and bumps. We particularly like setups such as the FUZE box, which embed the Pi into a more traditional, albeit chunkier, all-in-one device. Yes, the Raspberry Pi's tiny form factor is appealing, but with a heavy rotation of USB devices thrown in, it's all too easy to end up in cable-induced tiny hell.

Speaking of tiny things, to coincide with the new learning regimen, the BBC will distribute about a million 'Micro:bit' computers to Year 7 pupils. These are even smaller than the Pi, but have no means of output besides a 5x5 LED array. Unlike the Raspberry Pi, then, they cannot function as standalone computers, requiring instead to be programmed from a more capable device. Microsoft is generously providing the software and training for this venture, and we are relieved to hear it will not tie the product to its Windows 10 platform. The devices have gravity and motion sensors, as well as Bluetooth and some buttons. There are five GPIO rings that could connect further sensors or contraptions. They can be programmed in a number of languages including C++, Python, JavaScript and Blocks – a visual programming language. The Microsoft-provided code editors are all web-based, and code entered here must be compiled before being downloaded to the Micro:bit. At present, amidst the pre-launch hush (though it should be available in schools now), details are sketchy, but it seems as though this compilation all takes place on Microsoft servers. It'll be disappointing if the code editors can't be used offline or no option to compile locally is offered. You can find out more about it at **www.bbc.co.uk/mediacentre/mediapacks/microbit**.

Micro:bit spokesbods have been keen to stress that the device is in no way intended to compete with the Raspberry Pi, but rather to complement it. The Micro:bit features a 20-pin edge connector, which connects it to the Pi or another device, so that the machines can work in tandem. Such a pairing will be necessary for the Micro:bit to have a means of communicating with the outside world. The device – indeed the combination

»

» of the device and the Pi – has a great deal of potential but there exists some scepticism over whether anything like the excitement generated by the 1982 launch of BBC Micro will be seen again. Back then, computers were new and exciting, while these days kids expect them to provide a constant barrage of entertainment in the form of six-second cat videos or 140-character social commentary. Not all of them are going to be thrilled at having to program them. But anything that lowers, if not entirely obliterates, any entry barrier to getting into coding is fine by us. We also look forward to ne'er-do-well students hacking each other's Micro:bits or engaging them in a collaborative DDOS attack on their school's infrastructure.



› The Cortex M0 ARM processor is at the heart of the battery-powered Micro:bit.

Get with the program

The syllabus comprises three Key Stages. The first, for five- to six-year-olds, covers algorithms in a very general sense. Algorithms will be described in terms of recipes and schedules, to introduce the idea of formalising instructions. The second stage (ages 7 to 11) introduces core ideas, such as loops and variables. Alongside this, candidates will be learning to use web services and how to gather data. The final stage, for secondary students aged 11 to 14, requires students to learn at least two programming languages and understand binary arithmetic, Boolean algebra, functions and data types. Students will also touch on information theory, at least as far as how different types of information can all be represented as a string of bits. They will also gain insights into the relationship between hardware and software.

Throughout the curriculum, students will also learn the vital skills of online privacy and information security – skills the want of which has led to many an embarrassing corporate or governmental blunder. It's an ambitious project, but perhaps such a radical step is necessary to address the skills shortage in this area. With luck, the scheme will also lead to much-needed diversification among the coding populace. If it works out, and pupils are learning Python alongside Mandarin, or studying Kohonen or Knuth alongside Kant, then we'll be thrilled. ■

Code Clubs

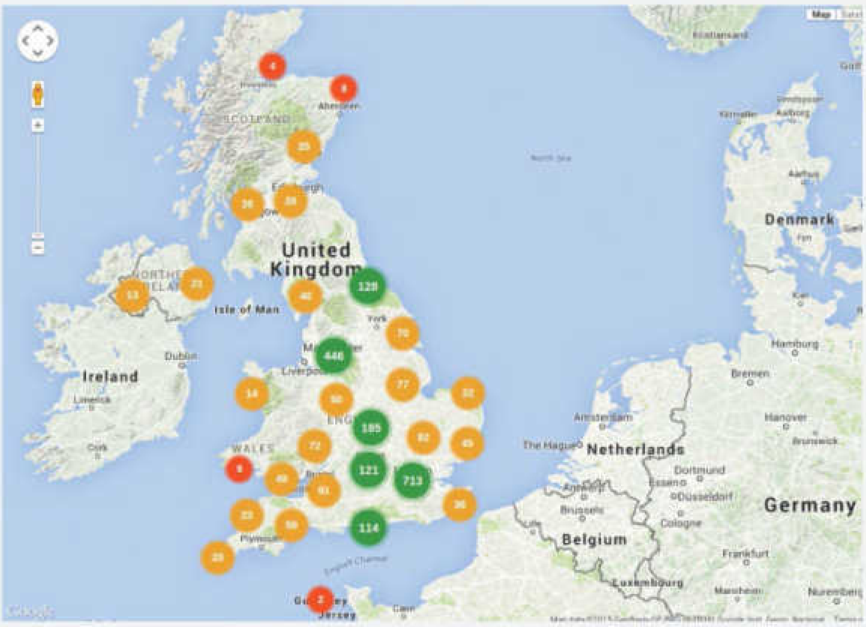
</code>

There are also over 2,000 volunteer-run code clubs across the UK. Code Club, armed with £100,000 courtesy of Google, provides the material, and schools (or other kind venues) provide space and resources for this noble extracurricular activity.

The Code Club syllabus is aimed at 9- to 11-year-olds and consists of two terms of Scratch programming, then a term of web design (HTML and CSS), concluding with a final term of grown-up Python coding. Projects are carefully designed to keep kids interested: they'll be catching ghosts and racing boats in Scratch, and doing the funky Turtle and keeping tabs on the latest Pokémon creatures in Python, to name but a few.

If you have the time and some knowledge, it's well worth volunteering as an instructor. All the tutorials are prepared for you and if you can persuade a local primary school to host you, you'll get a teacher to maintain order and provide defence against any potential pranksters. You will require a background check, though. Code Club also provides three specialist modules for teachers whose duty it is to teach the new Computing curriculum.

This kind of community thinking is very much in the spirit of open source, providing an access-for-all gateway to coding free from commercial interest and corporate control. The Code Club phenomenon is spreading worldwide, too. You'll now find them as far afield as Bahrain and Iceland. The project materials have already been translated into 14 languages, with more on the way. You can find out more at **www.codeclub.org.uk**.



72
techradar.

Get started with IDEs

Speed up your programming workflow by using Geany, a lightweight integrated development environment.

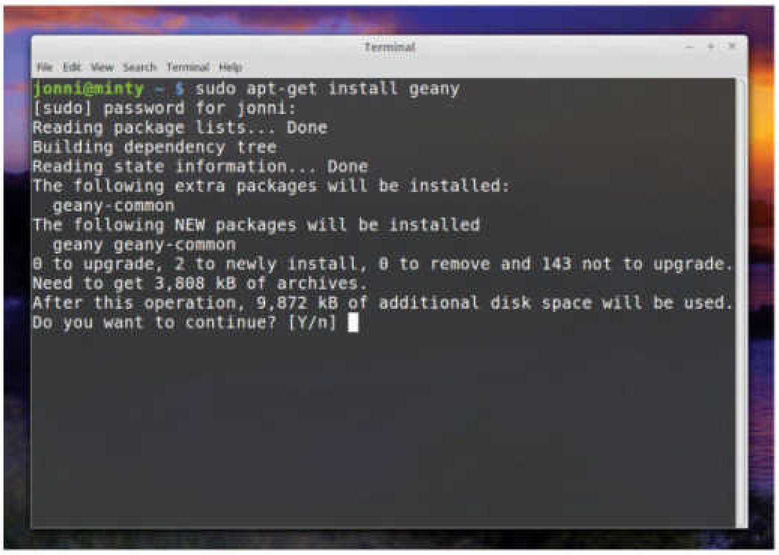
The cycle of editing, running and debugging even simple programs can rapidly become painful. So imagine how ugly things can get when working on a huge project involving complex build processes. This is why integrated development environments (IDEs) exist – so that the editing, compiling, linking and debugging can all take place on a unified platform (and you have a single point at which to direct programming rage). Some IDEs are tailored for a particular language, such as *IDLE* for Python *(see page 92)*. We're going to look at a more general purpose tool called *Geany*. *Geany* is pretty basic as far as IDEs go – in fact, by some measures, it's more of a text editor on steroids than an IDE, but it is more than sufficient for this tutorial.

You'll find *Geany* in Mint's repositories, so you can install it either from the Software application or by doing:

```
$ sudo apt-get update
$ sudo apt-get
install geany
```

Now start *Geany* from the Places menu. There's a quick run-through of the interface's major features opposite. We can start by opening up our helloworld.py program from before: Select File > Open and then navigate to our **python/ directory** to select the file.

> **"Variables allow us to abstract away specifics and deal with quantities that, well, vary."**



```
jonni@minty ~ $ sudo apt-get install geany
[sudo] password for jonni:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  geany-common
The following NEW packages will be installed
  geany geany-common
0 to upgrade, 2 to newly install, 0 to remove and 143 not to upgrade.
Need to get 3,808 kB of archives.
After this operation, 9,872 kB of additional disk space will be used.
Do you want to continue? [Y/n]
```

› Before you start, you need to install *Geany* from Mint's repositories.

The grown-up way to begin Python files is with a directive known as a shebang. This is a line that tells the shell how it ought to be run. So our Python 3 code should begin:

```
#!/usr/bin/env python3
```

The *env* program is a standard Linux tool that is always accessible from the **/usr/bin/** directory. It keeps track of where various programs are installed, because this can differ wildly from distro to distro. On Mint, our shebang could equally well call **/usr/bin/python3** directly.

Now we'll change up our program slightly. We're going to show how variables can be used as placeholders, and in doing so make our greeting a little more customisable. Edit the code (keeping the shebang above, unless you have some personal grievance) so that it looks like this:

```
name = 'Dave'
print('Hello ', name)
```

Note the space after `Hello`, otherwise the output would look a bit funny. We have declared a variable called `name` containing the string `'Dave'`. We can test our code by pushing F5 or clicking the Run button (see annotation). You probably knew what was going to happen, and you probably realise that the same effect could be achieved just by changing `'world'` to `'Dave'` in the original program – but there is something much more subtle going on here. Variables allow us to abstract away specifics and deal with quantities that, well, vary. We can elaborate on this concept a little more by introducing a new function, `input()`, which takes a line of user input and returns that string to our program. Change the first line to

```
name = input()
```

and run your code. In the terminal window, type your name, or whatever characters you can muster, and press Return. The program took your input and courteously threw it back at you. The `input()` function can also be given a string argument, which will be used to prompt the user for input. Our `print()` function above does our prompting for us, so we have no need for this.

Before we go further, it's worth talking about tabs and spaces. Python is terribly fussy about indentation, and even if it wasn't, you still should be. Correctly indenting code makes it easy to demarcate code blocks – lines of code that collectively form a function or loop – so we can see, for example, which level we are at in a nested loop. A tab

The Geany interface

Run button

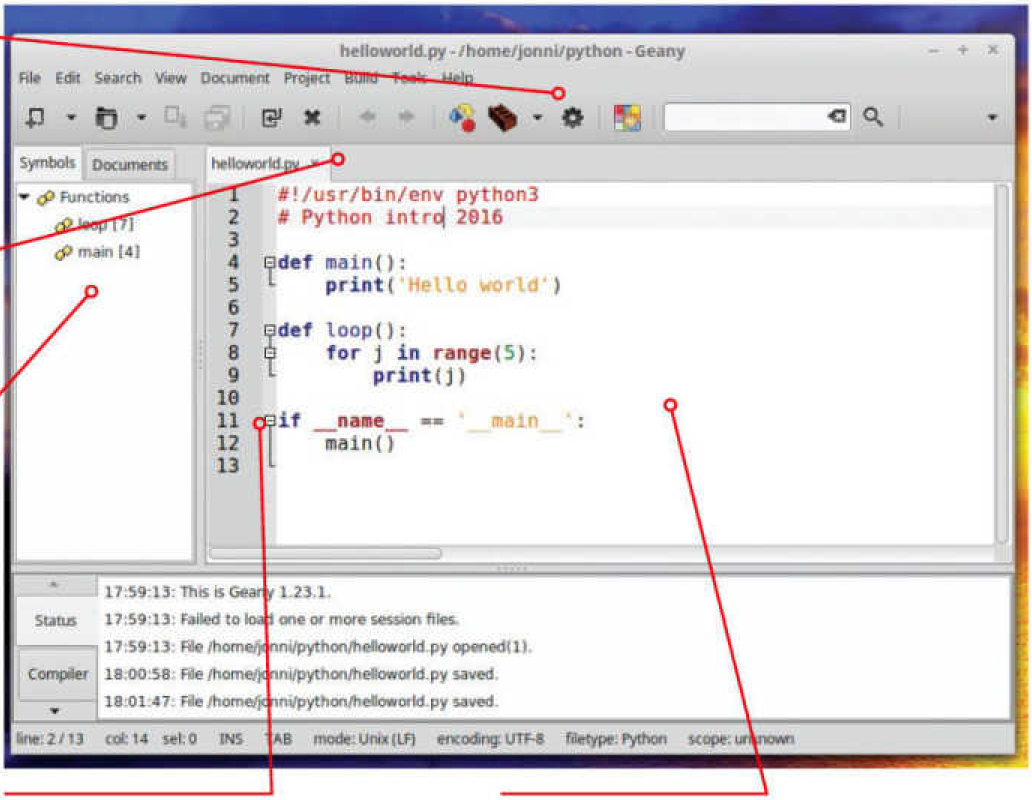
The Run button (or F5) will save and execute your Python program in a terminal. When the program finishes or breaks, you can study any output or error messages here before closing it.

Tabs

By default, Geany opens your recent files in tabs. So if you're working on a more involved, multi-file project, related code is just one click away.

Symbols

The Symbols tab lists all the functions, variables and imports in the current file, together with the line on which they are declared. Clicking will take you straight there.



```
#!/usr/bin/env python3
# Python intro 2016

def main():
    print('Hello world')

def loop():
    for j in range(5):
        print(j)

if __name__ == '__main__':
    main()
```

```
17:59:13: This is Geany 1.23.1.
17:59:13: Failed to load one or more session files.
17:59:13: File /home/jonni/python/helloworld.py opened(1).
18:00:58: File /home/jonni/python/helloworld.py saved.
18:01:47: File /home/jonni/python/helloworld.py saved.
```

line: 2 / 13 col: 14 sel: 0 INS TAB mode: Unix (LF) encoding: UTF-8 filetype: Python scope: unknown

Code folding

Code folding is de rigueur for any IDE. Clicking on the + or - will reveal or collapse the block of code on the adjacent line.

Code editor

The main area where you enter or edit code. Geany is good at guessing the language you're typing, and keywords are highlighted.

represents a fixed amount of white space decided by the text editor, but just because an editor renders a tab character as eight spaces, that doesn't make it equivalent to pushing the space bar eight times. Python happily works with files that use tabs or spaces, so long as they're consistent.

The reason Python is fussier than other languages about these matters is because indentation levels actually form part of the code. So just adding a random indent to your code is likely to break it. We need some more coding concepts at our disposal to illustrate this. Let's start with the `if` conditional. Suppose we want our program to behave differently when someone named, say, Agamemnon uses it. Replace the last line of our code with the following block, noting that *Geany* automatically indents your code after the `if` and `else` statements.

```
if name == ‘Agamemnon’:
      print(‘Ah, my old friend...)
else:
      print(‘Hello ’, name)
```

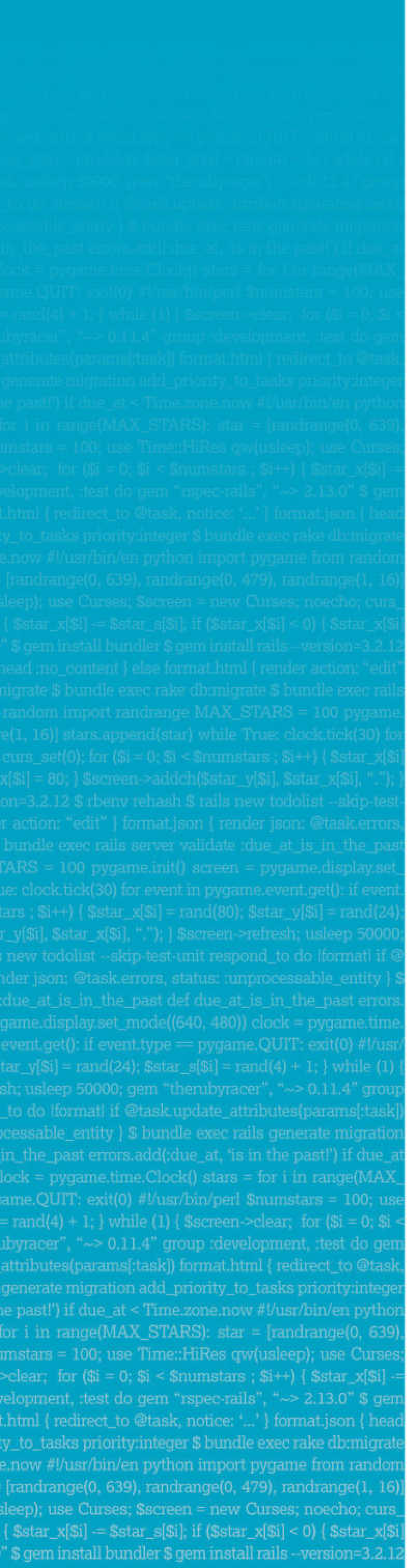
It can be hard to get used to the autotabbing, but it saves you considerable effort. The code we've just introduced is highly readable – we use the `==` operator to test for equality and use a colon to indicate a new code block is starting. If the condition is met (ie Agamemnon is visiting), then a special greeting is issued. We can put as many lines as we want in this clause – so long as they respect the indentation, Python knows that they all should be run when the condition is met. The `else` block is run when the condition is not met, – in this case, the program behaves exactly as it had done before. Note that strings are case-sensitive, so someone called agamemnon would not be recognised as an old acquaintance of our program. As well as testing for equality, we can also use the greater than (`>`) and less than (`<`) operators. The `len()` function returns the length of a string, so we could instead make our `if` statement:

```
if len(name) > 9:
      print(‘My what a long name you have’)
```

Now anyone whose name is 10 characters or more gets recognition. Of course, the `else` block from above can be included, too, if that behaviour is still desired.

There are many other IDEs besides *Geany*. If you want to check out something more advanced, see *PyCharm* or perhaps the PyDev plugin for *Eclipse*. Or maybe you're happier with a couple of terminals open – you wouldn't be the first. Also, don't forget *IDLE*, which is packaged with Python – we cover it in the Raspberry Pi chapter on page 92. However you choose to do it, though, we wish you good luck with your continued adventures in Linux and coding. ■

CODING
MADE SIMPLE



Coding basics

Now you have the tools, it's time to grasp the basics of coding

Get to grips with Python lists

Let's start by exploring some of the fundamental ideas behind programming logic, beginning with the concept of lists.

It's difficult to know where to start when it comes to explaining the basic programming concepts. When someone begins coding, there's an overwhelming number of different ideas to understand, absorb and eventually turn into solutions. And that's on top of the syntax and peculiarities of your chosen language.

One of the best approaches is to just start writing code, either by following an example project, such as the ones we provide in this book, or by piecing together examples from documentation. Both methods work, and they're the only real way to get to grips with the problems and complications of any particular concept.

We'd like to make this challenge slightly easier by looking at basic approaches that apply to the vast majority of circumstances and languages. And our first target is lists.

You don't get far with any moderately complex task without a list, whether that's a holiday to Diego Garcia or a trip to the supermarket. And list logic in code is just like in real life. A list is a convenient place to store loosely-related snippets of information. Programmers like to call loosely-related snippets of information a data structure, and a list is just another example. With a shopping list, for instance, the relationship might be something as simple as food – 'milk', 'bread' and 'baked beans'. If you were writing an application to remind you what's on your shopping list, it would make much more sense to put these values together. In Python, for example, you could create a list of those values with the following line:

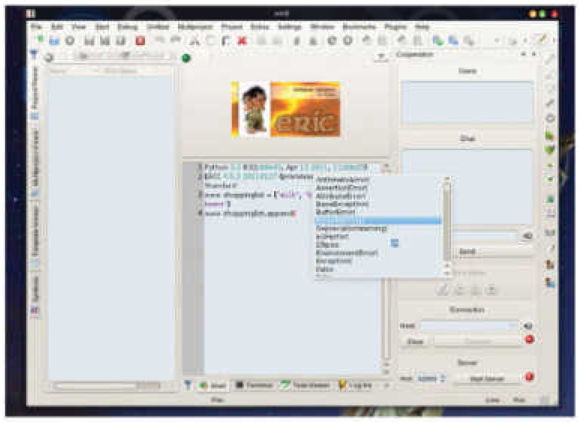
```
>>> shoppinglist = ["milk", "bread", "baked beans"]
```

This is technically a list of lists, because each value is itself a list – a string of characters. Some languages like to make that distinction, but most prefer the convenience.

Each of these items has been added to the freshly created **shoppinglist** list. But lists are dynamic, and one of the first things you often want to do is add new values.

This is where lists start to become interesting, because the method for adding and removing values can help to define the list's function. For example, it's faster for the CPU to add a new item to the end of a list because all the application has to do is link the last element to the new one.

If you insert an item into the middle of a list, the process first splits the links between the two items on either side of the insertion point, then links both the preceding item and the following one to the new item.



› If you're learning Python, the tab completion in the Eric programming environment is a great helper.

Stacks and queues

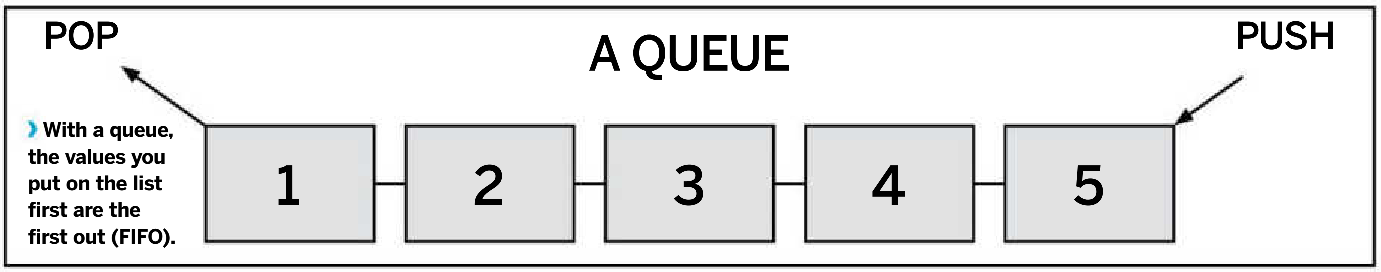
The most important characteristic for a list is that the chain of data it contains is in a specific order. It might not be important what that order is, but it's the order that differentiates a list from a random array of values, such as a chunk of memory.

It's for this reason that some of the earliest lists are stacks, with values either pushed on to the end or pulled off, rather than lists with values inserted and removed from the middle, which are more processor intensive.

Processing speed means that stacks aren't a necessity any more, but they're still useful for temporarily holding values, for example before retrieving them in reverse order like a stack of cards. The list could be your terminal command history, or a browser's 'Back' button.

In Python, you can execute a function called **append** to the same effect:

```
>>> shoppinglist.append("soup")
```



› With a queue, the values you put on the list first are the first out (FIFO).

This will add **soup** to our shopping list. When we've added it to our shop, it can be removed easily with **pop**:

```
>>> shoppinglist.pop()
```

If you're running these commands from the interpreter, you'll see the contents of the last value in the list as it's popped off the stack, in our case the word **soup**. If you've got processing limitations in mind, you might wonder why there isn't an option to pop the first value off the stack, too.

This would require the same amount of relinking as popping off a value from the end. It's perfectly possible, and turns a list into something called a 'queue' – the values you put on the list first are first out, known as FIFO. This is in contrast to a stack, which is LIFO.

There are many ways to use a queue, but the most obvious is to preserve the order of incoming data. As we explained, the only difference between a queue and a stack is where the **pop** value comes from, which is why Python doesn't have any special commands for accessing the first value, it just uses an argument for the **pop** command. The following will remove the first item in the list, and should output 'milk' from the interpreter:

```
>>> shoppinglist.pop(0)
```

Modern lists

Lists are no longer limited by processor speed to stacks and queues, and most modern languages and frameworks provide a vast number of handy functions for dealing with them. These offer a big advantage over the original primitive operations, and can turn lists into super-flexible data types.

Python is particularly good at this, and includes plenty of built-in functions for manipulating lists without having to write your own code. You can output a value from any point in the list by treating it as an array. Typing **shoppinglist[1]**, for instance, will return the second value in the list. As with nearly all things code-wise, **0** is the real first element in a list and array, making **1** the second. And if you want to output the entire list, you typically need to construct a simple loop that shuffles through these values from beginning to end.

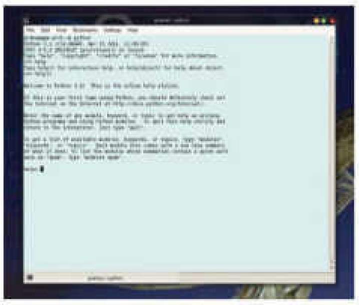
Using Python's interpreter

You might wonder why we have three **>** symbols preceding the snippets of code on these pages, as well as in our more comprehensive Python tutorials.

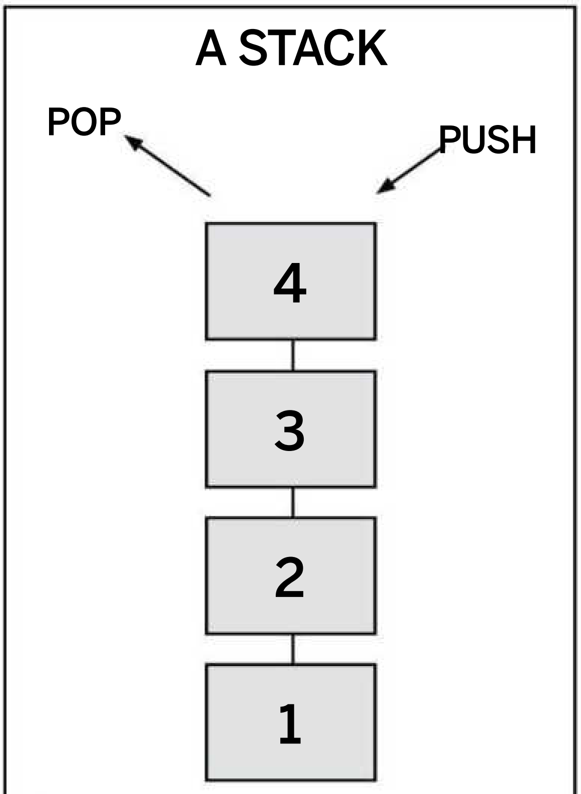
These symbols represent the cursor of Python's interactive interpreter, and they help to distinguish between something you might want to build into a script and something you can try immediately by typing in to the interpreter – although they are, in fact, interchangeable.

Launching the interpreter is as simple as typing **python** from the command line without any further arguments. It's at this point you'll see the cursor symbols, and you can now start typing lines of Python code.

This is a brilliant way of learning how the language works, as well as experimenting with syntax and the concepts you're working with. Errors in a single line are far easier to spot than those in a 20-line script that's being run for the first time, and you can also get a really good feel for how Python, and programming in general, handles logic and data. If you execute a function that returns a value, for example, such as **len(list)** to return the length of a list, the value is output to your terminal immediately, rather than needing any explanation in a script or assigning to another variable. You can quit the interpreter by pressing [Ctrl]+[D] together, but you will obviously lose the current state of your code.



If you're learning to program, the Python interpreter can teach you a great deal, regardless of your chosen language.



With a stack, the values you put on the list last are the first out (LIFO).

Python is an exception, because simply typing **shoppinglist** will output the list's contents, but for other languages, you might have to construct something like:

```
>>> for x in range(0,len(shoppinglist)):
...     shoppinglist[x]
...     print
```

The **for** loop here gives a value to **x** that steps from **0** to **len(shoppinglist)**, which is a method to return the length of a list. Each location in the list is output as we loop through it in the order they were added (FIFO).

You can change the order in lots of ways, but Python's **sort** method is tough to beat:

```
>>> shoppinglist
['baked beans', 'bread', 'milk', 'soup']
>>> shoppinglist.append("apples")
>>> shoppinglist
['baked beans', 'bread', 'milk', 'soup', 'apples']
>>> shoppinglist.sort()
>>> shoppinglist
['apples', 'baked beans', 'bread', 'milk', 'soup']
```

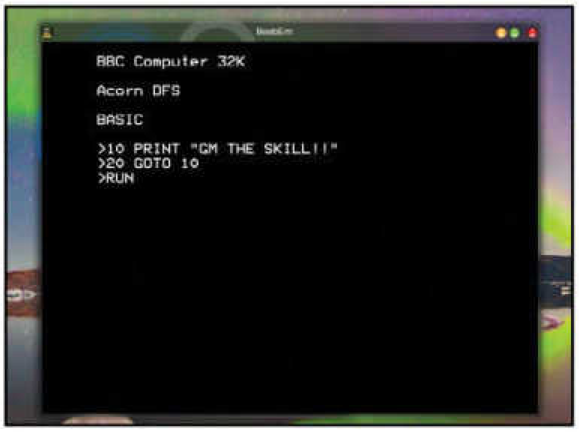
If you can follow the logic of that simple piece of code, then it's safe to assume you've grasped the logic of lists. There's really nothing more to it, and now we hope you're wondering what all the fuss is about. The main problem is that it can get complicated quickly, and because lists are shorthand for so many different data types, it can be difficult working out what might be happening in any one chunk of code. This is especially true of Python – because it has so many convenience functions, unless you know what they do, you're unlikely to grasp the logic. But at their heart, they're just a chain of values, and that's the best place to start. ■

Quick tip

We're using Python for our examples because it's a great language for beginners. But every concept we discuss works with other languages, too – it's just a matter of finding the correct syntax for the functionality.

Understanding functions & objects

After introducing the concept of lists, it's time for us to tackle the fundamental building blocks behind any application.



```
BBC Computer 32K

Acorn DFS

BASIC

>10 PRINT "GM THE SKILL!!"
>20 GOTO 10
>RUN
```

› BBC BASIC was the first programming language many of us came into contact with.

We've already covered a concept called a list. Lists have been around since people started to write one line of code after another because, in essence, they simply store one value after another.

But what must have come soon after lists in the mists of language creation is the idea of a function, because a function is a kind of list for code logic. It's not built at runtime, but while you're constructing your project, and much like their mathematical counterparts, functions are independent blocks of code that are designed to be reused.

Functions are fundamental for all kinds of reasons. They allow the programmer to break a complex task into smaller chunks of code, for instance, and they allow you to write a single piece of code that can be reused within a project. You can then easily modify this code to make it more efficient, to add functionality or to fix an error. And when it's one piece of code that's being used in lots of places, fixing it once is far easier than trawling through your entire project and fixing your same broken logic many times.

Functions also make program flow more logical and easier to read, and you can keep your favourite functions and use them again and again. When you break out of single-file projects, functions become the best way of grouping functionality and splitting your project into different source files, naturally breaking complex ideas into a group of far easier ones. If you then work with a team of programmers, they can work on functions and files without breaking the operation of the application, and it's this idea that led to the creation of objects. These are a special kind of function that bundle both the code logic and the data the logic needs to work into completely encapsulated blocks. But let's start at the beginning. If your first programming project was from about 30 years ago, and written on a home computer, then there's a good chance it was written in BASIC – a language embedded within the ROMs of many of the home computers of the early eighties.

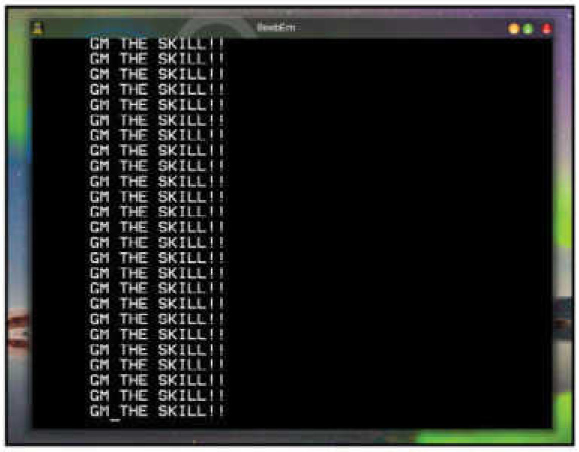
BASIC code was mostly sequential, with many systems even using line numbers, and there's a good chance that the following might have been your first program:

```
10 PRINT "Hello world!"
```

This led to one of the most primitive hacks of the day. You had to find a computer shop displaying a selection of machines. You'd then sneak in and quickly type the above line followed by **20 GOTO 10**, **RUN** and a carriage return. This would create an infinite loop that printed 'Hello World' for ever – or for the 10 minutes it took for the store staff to notice and wearily turn it off and on again. The cooler kids would change the text to say something rude and add colour to the output. Either way, it's a basic example of a function if you can execute it from another part of your project.

BASIC programming without functions can be a little like writing assembly language code. If you want to repeat a series of lines, you need to remember where you are in the code, jump to the new code, and jump back after execution. BASIC provided some of this functionality with procedures, but functions give that chunk of code a label, which can then be called from any other point in your project. Functions can also return a value, such as the sum of a group of numbers, and accept arguments. That could be a list of numbers to be added, for example. But you'd have needed to be a wealthy Acorn Archimedes owner to get those facilities out of BASIC.

With Python, you can try these ideas without getting too retro. From the interpreter, for example (just type **python** on the command line and add our code), type the following:



```
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM THE SKILL!!
GM_THE SKILL!!
```

› BASIC wasn't always put to the best uses.

```
>>> def helloworld():
...     print (“Hello World”)
...     return
...
```

As ever with Python, you need to be very careful about formatting. After creating a new function called **helloworld()** with the **def** keyword, the interpreter precedes each new line with **...**. Any lines of code you now add will be applied to the function, and you'll need to press [Tab] at the beginning of each to indent the code and make sure Python understands this. Other languages aren't so picky. After the **return** statement, which signals the end of the function, press [Return] to create a blank line. You've now created a function, and you can execute it by typing **helloworld()** into the interpreter. You should see the output you'd expect from a function with a name like **helloworld()**, and you've just jumped forward 30 years.

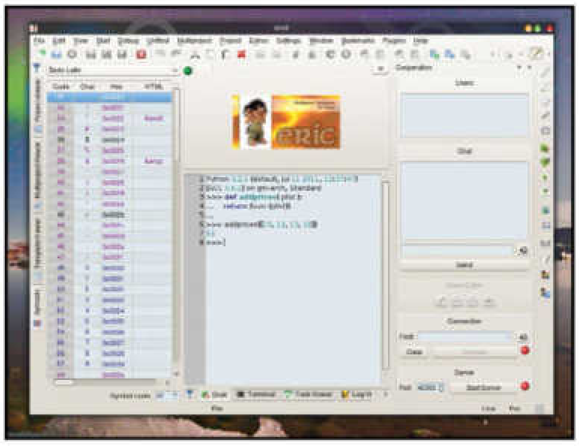
Passing arguments

There's a problem with our new function: it's static. You can't change anything. This can be solved by opening the function up to external influence, and you can do that by enabling it to pass and process arguments. If you alter the function definition to **def helloworld(str)**, for instance, Python will expect to find a string passed to the function whenever it's called. You can process this string within the function by changing **print (“Hello World”)** to **print str**, but you should obviously be doing anything other than simply printing out what's passed to the function. It can now be executed by typing **helloworld(“something to output”)** and you'll see the contents of the quotes output to the screen.

To show how easy it is to create something functional, we'll now build on the idea of lists from the previous tutorial. Instead of a shopping list, we'll create a list of prices and then write a function for adding them together and returning the total value. Here's the code:

```
>>> def addprices( plist ):
...     return (sum (plist))
...
>>> pricelist = [1.49, 2.30, 4.99, 6.50, 0.99]
>>> addprices(pricelist)
16.27
```

Python is very forgiving when it comes to passing types to functions, which is why we didn't need to define exactly what **plist** contained. Most other languages will want to know exactly what kind of types an argument may contain before it's able to create the function, and that's why you often get definitions before the program logic, and why types are often declared in header files because these are read before the main source code file.



› An IDE, such as *Eric*, can make working with functions and objects easier.

Working out what your language requires and where is more difficult than working out how to use functions. And that's why they can sometimes appear intimidating. But Python is perfect for beginners because it doesn't have any convoluted requirements – apart from its weird tab formatting. All we've done in our code is say there's going to be some data passed as an argument to this function, and we'd like Python to call this **plist**. It could even have had the same name as the original list but that might look confusing.

The only slight problem with this solution is that you'll get an error if you try to pass anything other than a list of numbers to the function. This is something that should be checked by the function, unless you can be certain where those numbers come from. But it does mean you can send a list of values in exactly the same way you define a list:

```
>>> addprices([10, 12, 13, 18])
53
```

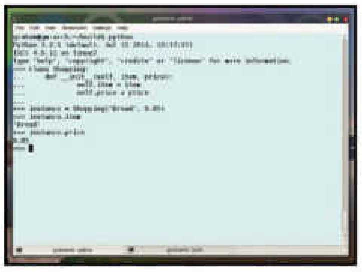
The biggest challenge comes from splitting larger problems into a group of smaller ones. For a programmer, this task becomes second nature, but for a beginner, it's an intimidating prospect. The best way is to first write your application, regardless of functions or programming finesse. Often, it's only when you're attempting to write your first solution that better ones will occur to you, and you'll find yourself splitting up ideas as the best way to tackle a specific issue within your project. The best example is a GUI, where it's common sense to give each menu item its own function, which can be launched from either the icons in the toolbar or by selecting the menu item. Many developers consider their first working copy to be a draft of the final code, and go to the trouble of completely rewriting the code after proving the idea works. But don't worry about that yet. ■

The art of objects

An object bundles functions and the type of data they accept together, so that there's less ambiguity on how a function should be used, and better separation from the main flow and side processes. Dealing with objects can quickly get complicated, partly because there are so many different concepts, language-specific features and implementations, and partly because it requires the programmer to think about problems differently. This simple example creates a class that contains both the **item** and the **price** from our previous shopping list:

```
>>> class Shopping:
...     def __init__(self, item, price):
...             self.item = item
...             self.price = price
...
>>> instance = Shopping(“Bread”, 0.85)
>>> instance.item
‘Bread’
>>> instance.price
0.85
```

A class is a bit like the specification for an object, whereas the term **object** refers to when specification is used within your code. In our example, the class is the structure of **Shopping** while the object is **instance**. The function called **__init__** is run when the object is created, and we pass the two values – one for the item's name and the other for its price. You should be able to see how this class could be quickly expanded to include other functions (or methods), and how the world of classes and objects can transform your coding projects for the better.



› You can still create classes and functions from within the interpreter, but it begins to make less sense.

Quick tip

When you're using other packages, only the functions are exposed. You don't need to know how they work, just what input they're expecting and what output they return. That's functional programming in a nutshell.

Adapt and evolve using conditionals

Any non-trivial program needs to make decisions based on circumstances. Get your head around coding's ifs and buts.

Murray Walker, the great Formula 1 commentator, used to say "IF is F1 spelled backwards." Ifs, buts and maybes play a vital role in motor racing, as they do in computer programming. If you're writing a program that simply processes and churns out a bunch of numbers, without any user interaction, then you might be able to get away without any kind of conditional statements. But most of the time, you'll be asking questions in your code: if the user has pressed the [Y] key, then continue. If not, stop. Or if the variable PRICE is bigger than 500, halt the transaction. And so forth.

A condition is just a question, as in everyday life: if the kettle has boiled, turn off the gas. (We don't use newfangled electricity around here.) Or if all the pages have gone to the printers, go to the pub. Here's an example in Python code:

```
x = 5

if x == 10:
      print "X is ten"
else:
      print "X is NOT ten"

print "Program finished"
```

Here, we create a new variable (storage place for a number) called **X**, and store the number **5** in it. We then use an **if** statement – a conditional – to make a decision. If **X** contains **10**, we print an affirmative message, and if not (the **else** statement), we print a different message. Note the double-equals in the **if** line: it's very important, and we'll come on to that in a moment.

Here, we're just executing single **print** commands for the **if** and **else** sections, but you can put more lines of code in there, providing they have the indents, Python style:

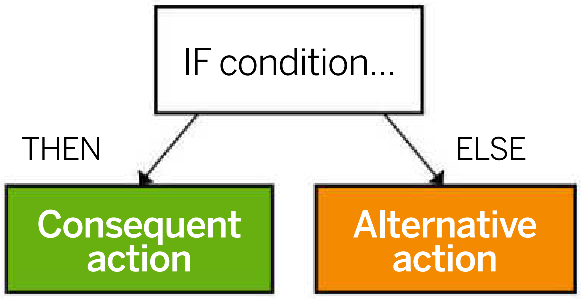
```
if x == 10:
            print "X is ten"
            somefunction()
else:
            anotherfunction()
```

PEP: 3103
Title: A Switch/Case Statement
Version: e9c1b9aa2108
Last-Modified: 2008-12-16 09:03:23 +0000 (Tue, 16 Dec 2008)
Author: guido at python.org (Guido van Rossum)
Status: Rejected
Type: Standards Track
Content-Type: text/x-rst
Created: 25-Jun-2006
Python-Version: 3.0
Post-History: 26-Jun-2006

Contents

- Rejection Notice
- Abstract
- Rationale
- Basic Syntax
  - Alternative 1

› **Python doesn't have switch/case. Why? Read the explanation at www.python.org/dev/peps/pep-3103/**



› **At its core, a conditional statement is something like this.**

In this case, if **X** contains **10** we print the message as before, but then call the **somefunction** routine elsewhere in the code. That could be a big function that calls other functions and so forth, thereby turning this into a long branch in the code.

There are alternatives to the double-equals we've used:

» **if x > 10**	If X is greater than 10
» **if x < 10**	If X is less than 10
» **if x >= 10**	If X is greater than or equal to 10
» **if x <= 10**	If X is less than or equal to 10
» **if x != 10**	If X is NOT equal to 10

These comparison operators are standard across most programming languages. You can often perform arithmetic inside the conditional statement, too:

```
if x + 7 == 10:
      print "Well, X must be 3"
```

Comparing function results

While many **if** statements contain mathematical tests such as above, you can call a function inside an **if** statement and perform an action depending on the number it sends back. Look at the following Python code:

```
def truefunc():
      return 1

def falsefunc():
      return 0

print "Execution begins here..."

if truefunc():
      print "Yay"

if falsefunc():
      print "Nay"
```

The first four lines of code define functions (subroutines) that aren't executed immediately, but are reserved for later use. They're really simple functions: the first sends back the number 1, the second zero. Program execution begins at the **print** line, and then we have our first **if** statement. Instead of doing a comparison, we call a function here, and act on the result. If the **if** statement sees the number **1**, it goes ahead with executing the indented code, if not, it skips past it. So when you run this, you'll see **Yay** but not **Nay**, because **if** here only does its work if it receives the number **1** back from the function it calls. In Python and many other languages, you can replace **1** and **0** with **True** and **False** for code clarity, so you could replace the functions at the start with:

```
def truefunc():
      return True

def falsefunc():
      return False
```

and the program will operate in the same way.

Clearing up code

In more complicated programs, a long stream of **ifs** and **elses** can get ugly. For instance, consider this C program:

```
#include <stdio.h>

int main()
{
            int x = 2;

            if (x == 1)
                        puts("One");
            else if (x == 2)
                        puts("Two");
            else if (x == 3)
                        puts("Three");
}
```

C, and some other languages, include a **switch** statement which simplifies all these checks. The lines beginning with **if** here can be replaced with:

```
            switch(x) {
                        case 1: puts("One"); break;
                        case 2: puts("Two"); break;
                        case 3: puts("Three"); break;
            }
```

This is neater and easier to read. Note that the **break** instructions are essential here – they tell the compiler to end the switch operation after the instruction(s) following **case** have been executed. Otherwise, it will execute everything following the first matching case.

Another way that some programmers in C-like languages shorten **if** statements is by using ternary statements. Consider the following code:

```
if (x > y)
            result = 1;
else
            result = 2;
```

This can be shortened to:

```
result = x > y ? 1 : 2;
```

Here, if **X** is bigger than **Y**, result becomes **1**; otherwise it's **2**. Note that this doesn't make the resulting code magically smaller – as powerful optimising compilers do all sorts of tricks – but it can make your code more compact.

So, that's conditionals covered. Although we've focused on Python and C in this guide, the principles are applicable to nigh-on every language. And it all boils down to machine code at the end of the day – the CPU can compare one of its number storage places (registers) with another number, and jump to a different place in the code depending on the result. Remember to thank your CPU for all the hard work it does. ■

Assignment vs comparison

In some languages, especially C, you have to be very careful with comparisons. For instance, look at this bit of C code – try to guess what it does, and if you like, type it into a file called **foo.c**, run **gcc foo.c** to compile it and then **./a.out** to execute it.

```
#include <stdio.h>

int main()
{
   int x = 1;

   if (x = 5)
            puts("X is five!");
}
```

If you run this program, you might be surprised to see the **X is five** message appear in your terminal window. Shurley shome mishtake? We clearly set the value of **X** to be **1**! Well actually, we did, but in the **if** line we then performed an assignment, not a comparison. That's what the single equals sign does.

We're not saying "if X equals 5", but rather, "put 5 in X, and if that succeeds, execute the code in the curly brackets". Ouch. If you recompile this code with **gcc -Wall foo.c** (to show all warnings), you'll see that **GCC** mentions **assignment used as truth value**. This is an indication that you might be doing something wrong.

The solution is to change the **if** line to **if (x == 5)** instead. Now the program runs properly. It's a small consideration, but if you've just written a few hundred lines of code and your program isn't behaving, this could be the root cause. It's caught us out many times...

Big ifs and small ifs

In C, and other languages that share its syntax, you don't need to use curly brackets when using an **if** statement followed by a single instruction:

```
if (a == 1)
   puts("Hello");
```

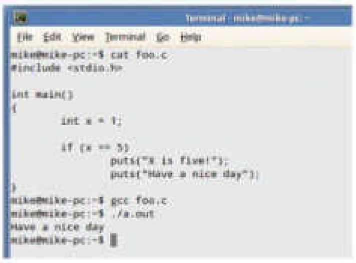
But only one instruction will be executed. If you do something like this:

```
int x = 1;

if (x == 5)
   puts("X is 5");
   puts("Have a nice day");
```

When you execute it, it won't print the first message, but it will print the second. That's because only the first is tied to the **if** statement. To attach both instructions to the **if**, put them in curly braces like the following:

```
if (x == 5) {
   puts("X is 5");
   puts("Have a nice day");
}
```

Contrast this with Python, where the indentation automatically shows which code belongs to which statement. C doesn't care about indentation – you have to explicitly show what you want with curly brackets.



```
mike@mike-pc:~$ cat foo.c
#include <stdio.h>

int main()
{
        int x = 1;

        if (x == 5)
                puts("X is five!");
                puts("Have a nice day");
}
mike@mike-pc:~$ gcc foo.c
mike@mike-pc:~$ ./a.out
Have a nice day
mike@mike-pc:~$
```

Here we see how indentation isn't good enough for C – we need to use curly braces to bundle code.

Variable scope of various variables

Caution! That variable you're accessing might not be what you think it is. Why? Because variables are variable. Have we said 'variable' enough now?

Once upon a time, programs were just big lists of instructions, and these instructions could do anything with variables and the contents of memory. This was fine for simple, low-level programs but as time went on and people starting making bigger and more complicated programs, the idea that one part of the program could do anything with memory became less appealing.

For instance, you might have a program that controls a robotic arm, and stores the state of the arm in a variable called **X**. Your program calls lots of different subroutines, some of which may have been programmed by other people. How do you know that those routines aren't messing about with the RAM where **X** is stored?

The results could be disastrous, especially when the arm starts punching people instead of stroking puppies as originally envisaged.

This is where the concept of variable scope comes into play. In programming, scope defines how a variable is visible to the rest of the program. In most high-level languages, we can control the scope of a variable – that is, we can choose whether it should be visible to the current chunk of code, to the current source code file, or absolutely everywhere in all of the program's files.

This level of control is absolutely fundamental to good program design, so that multiple programmers can work together on a project, implementing their own routines without accidentally trampling over one another's data. It's good for modularisation and keeping data safe and secure, and therefore it's a good thing to get right in the early days of your programming journey.

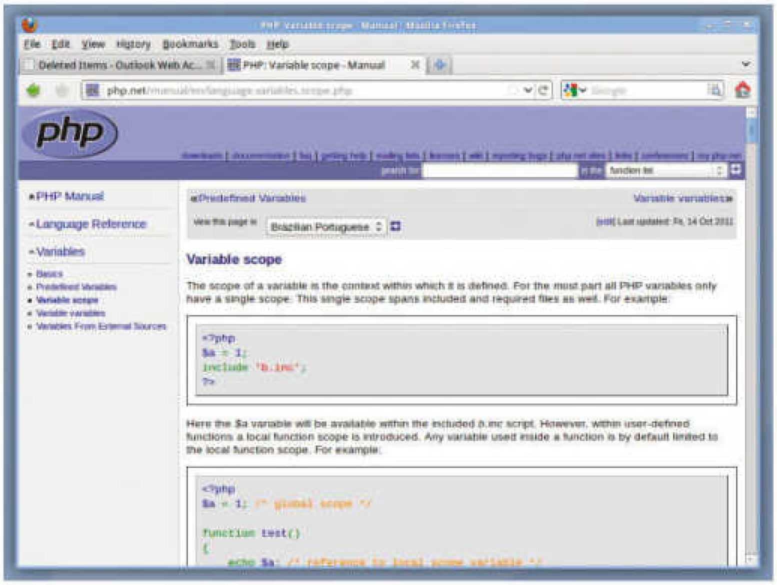
Let's start with a bit of Python code. Try this:

```
def myfunc():
      x = 10
      print x


x = 1
print x
myfunc()
print x
```

If you're totally new to Python: the **def** bit and the following two lines of code are a function which we call later. Program execution begins with the **x = 1** line. So, we create a variable called **x** and assign it the number **1**. We print it out to confirm that. We then call a function which sets **x** to **10**, and prints it. As control jumps back to the main chunk of our code, we print **x** again… and it's back to **1**. How did that happen? Didn't we just set **x** to **10** in the function?

Well, yes, but the function had its own copy of the variable. It didn't do anything with the **x** variable that was declared outside of it. The function lives happily on its own, and doesn't want to interfere with data it doesn't know about. This is essential to keeping code maintainable – imagine if all variables were accessible everywhere, and you were writing a routine to be dropped inside a 50,000-line software project. You'd be totally terrified of choosing variable names that might be in use elsewhere, and accidentally changing someone else's data.

› Most well-documented languages have explanations of variable scope, so once you've got the basics from this article, you can explore specific implementations.


PHP: Variable scope - Manual - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Deleted Items - Outlook Web Ac...
PHP: Variable scope - Manual
php.net/manual/en/language.variables.scope.php
php
search for
in the
function list
PHP Manual
Language Reference
Variables
Basics
Predefined Variables
Variable scope
Variable variables
Variables From External Sources
«Predefined Variables
Variable variables»
view this page in
Brazilian Portuguese
[edit] Last updated: Fri, 14 Oct 2011

Variable scope

The scope of a variable is the context within which it is defined. For the most part all PHP variables only have a single scope. This single scope spans included and required files as well. For example:

```
<?php
$a = 1;
include 'b.inc';
?>
```

Here the $a variable will be available within the included b.inc script. However, within user-defined functions a local function scope is introduced. Any variable used inside a function is by default limited to the local function scope. For example:

```
<?php
$a = 1; /* global scope */

function test()
{
    echo $a; /* reference to local scope variable */
```

Change of routine

So most programming languages provide this level of protection. Still, there are legitimate reasons why you might want to make a variable accessible to other routines, and in Python you can do this by inserting the following line into the start of the **myfunc()** routine:

```
global x
```

This makes all the difference. Previously, the version of **x** we were using in **myfunc()** was a local variable – that is, it affects only the code located inside the function. By adding the **global** command, we access **x** as a global variable – that is, it's the same one as we used in the main body of the code. So if we run this Python code now, we see that **x** is set to **1** at the start, but then the **myfunc()** routine grabs that **x** as a global variable and sets it to **10**, and it stays that way back in the main code.

How variable scope is handled varies from language to language, so let's look at another implementation here, this time in C. Whereas Python is pretty flexible about using variables that you haven't defined previously, C is a little more strict. Consider this:

```
#include <stdio.h>

void myfunc();

int main()
{
        int x = 10;
        printf(“x is %d\n”, x);
        myfunc();
        printf(“x is %d\n”, x);
}

void myfunc()
{
        int x = 20;
        printf(“x is %d\n”, x);
}
```

Here, because we're explicitly creating variables with **int**, it's somewhat clearer that **myfunc()** has its own version of the variable. But how do you go about making the variable global in this particular instance? The trick is to put the variable declaration outside of the functions. Move **int x = 10;** from inside **main()** to after the **void myfunc();** line at the top, and now the variable has become global – that is, it's accessible by all functions in the file.

However, that's not the only job you need to do. Inside the **myfunc()** routine, we still have a line that says **int x = 20;**. The **int** here is still creating its own version of the variable, so take it away so that the line just reads **x = 20;**. Run the program, and you'll see the intended result, that **x** is global and can be modified everywhere.

Note that while putting the **x** declaration outside of functions makes it global to the current source code file, it doesn't make it global absolutely everywhere in all of the source code. If you've got a big project with multiple C files, you'll have to use the **extern** keyword. So if you have **int x = 10;** in **foo.c** and you want to use that in **bar.c**, you'd need the line **extern int x;** in the latter file.

All of this leads to one final question: when should you use global variables, and when local? Ultimately, that's a matter of personal choice for your particular program, but the general rule is: use global variables only when necessary. Try to keep data isolated inside functions, and only use a global variable when you have no other option.

There are usually ways to access the local variables from one function in another (for example, pointers in C and references in C++), but those features are specific to individual languages, so that's something to look up in your own time. Pointers are a tricky business! ■

Automatic variables

Most local variables are automatic – that is, they are only created in RAM when program execution reaches the point of their initialisation. Once the routine containing the local variable has finished, that variable will no longer be useful, because outside routines can't access it. So, the compiler will typically reclaim memory used by that variable, making room for other bits and bobs.

There is, however, a way to stop that, and that's by declaring a static variable. This is still a local variable that can be manipulated only by code inside the current function, but it retains its state every time the function is called. For instance, look at the following C code:

```
#include <stdio.h>

void myfunc();

int main()
{
        myfunc();
        myfunc();
        myfunc();
}

void myfunc()
{
        int x = 0;
        x = x + 1;
        printf(“x is %d\n”, x);
}
```

Here, we call **myfunc()** three times, which creates a variable **x** containing **zero**, adds **1** to it and prints it out. You've no doubt already guessed that running this produces **1** with each call of **myfunc()**, because **x** is being created each time. But what happens if you change the declaration in **myfunc()** to **static int x = 0;**? Well, the output becomes this:

```
x is 1
x is 2
x is 3
```

The **static** keyword here tells the compiler that it should preserve the state of the variable between calls. So on the very first call it sets up **x** as **zero**, but in future calls it doesn't initialise the variable as new again, but retrieves its state from before. The compiler doesn't ditch it at the end of the function, as it does with automatic variables. So, using static variables is a great way to get some of the benefits of global variables (retaining its state throughout execution) without exposing it to the rest of the program.

Remember to initialise!

By default, most C compilers don't set newly-created automatic variables to zero. This is especially true in the case of automatic variables, because they might be created somewhere in RAM that was used previously by other data.

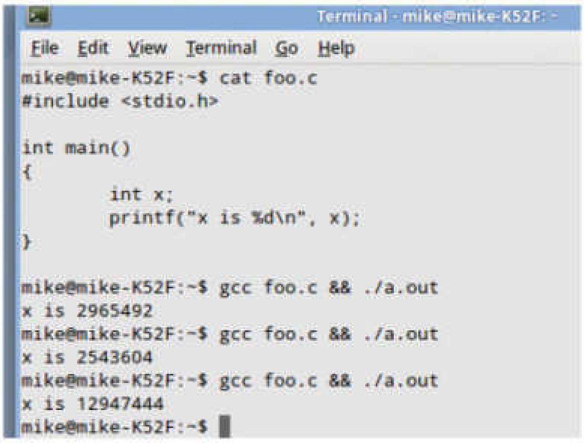
So if you have a program like this:

```
#include <stdio.h>

int main()
{
        int x;
        printf(“x is %d\n”, x);
}
```

then compile and run it, and then compile and run it again, the number will probably be different each time you do. The moral of the story is: don't blindly use variables that haven't been initialised with some kind of value!

Now, why don't compilers just set the variable to zero? That would make everything a lot safer, right? Well, yes, but it's extra work for the CPU and adds a performance hit. The compiler allocates space for variables in RAM that may have been used for other data before, so there's no guarantee that the RAM is zeroed-out.

Setting that chunk of RAM to zero requires an extra CPU instruction, and *GCC* tries to be pretty efficient. You can tell *GCC* to warn about the use of uninitialised variables with the **-Wall** flag.



```
mike@mike-K52F:~$ cat foo.c
#include <stdio.h>

int main()
{
        int x;
        printf("x is %d\n", x);
}

mike@mike-K52F:~$ gcc foo.c && ./a.out
x is 2965492
mike@mike-K52F:~$ gcc foo.c && ./a.out
x is 2543604
mike@mike-K52F:~$ gcc foo.c && ./a.out
x is 12947444
mike@mike-K52F:~$ 
```

› If you try to use automatic variables that haven't been given any value, prepare for some horrors.

Building proper programs

Learn to break problems down into manageable chunks, as we show you how to design a well-sculpted program.

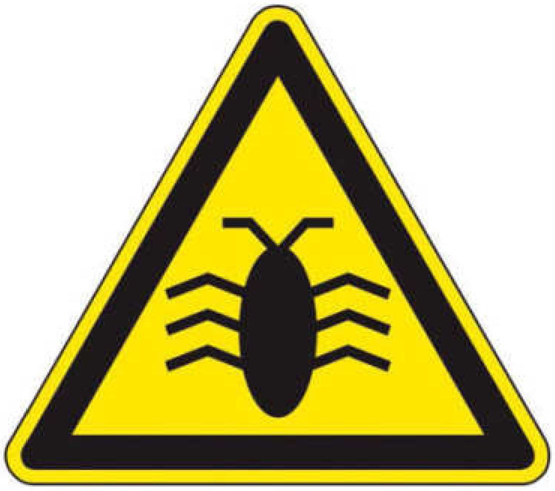
In this article, we're going to do something a little different. Instead of looking at a specific feature that shows up in many programming languages, we're going to look at the art of designing programs.

Knowing about **if** clauses, **for** loops, lists and functions (and much more) is vital if you want to learn to program. But you can get to a point where you know all of the elements, but you don't have the faintest clue where to get started when faced with a new programming challenge.

There are, however, some set processes – recipes of a kind – that you can follow that will help to direct your thinking and make solving a problem much easier.

Some of the Python features we use to solve problems might be unfamiliar, but that's not really the important thing in this tutorial (you can always look those up on the internet yourself) – the important thing is the thought process that we must go through.

To demonstrate, we need an example problem, and we've settled on a simple mortgage calculator. Let's say that Mike needs a mortgage of £150,000 to buy a nice house in Innsbruck. A bank has agreed to loan him the money at a rate of 6% per annum, with Mike paying it back over 25 years at a fixed monthly rate. Another bank agrees to lend him the money, but at 4% per annum, paying it back over 30 years. Mike wants to know what his monthly repayments will be in each situation, and how much interest he'll have paid in the end. He wants us to help him. The first step when tackling a programming problem is to closely examine the specification. Read through the above text carefully and pick out all the types of data that are described. As you're doing this, separate the data into input and output.



› Check often for bugs in your program. (Picture from: www.flickr.com/photos/fastjack/282707058)

Output data: monthly payments and total interest
Input: mortgage value, annual interest rate, duration

Great, that was pretty easy; but what was the point? Well, take a look at the output data; there are two types of data there. If we have to come up with two pieces of output data, that sounds like two problems instead of one – sneaky!

You'll find that many programming problems look like this – you'll start with a grand description of what you want to do: "I want to make Space Invaders," but on closer inspection you'll see that the problem's actually made up of many smaller ones.

Identifying these smaller issues, by looking at the data involved, is often the first step to finding a solution. This is called top-down development.

Functions: one task at a time

Once you've identified the smaller problems, your next task will be to focus in on one. We're going to start with the monthly payments problem.

If you look back at the tutorial on page 10 of this bookazine, you'll see us explaining that functions are great because they allow you to break a complex task into smaller, easier ones. Since that sounds exactly the same as what we're doing now, let's try to create a function for our monthly payments problem.

```
def monthlyPayments(mortgage, interest, term):
        monthlyPayments = 0
        return monthlyPayments
```

There's not much to this function yet; it's more of a sketch than something that might be useful, but it's often helpful to start with a sketch and fill it out later. Let's take a look and see what we have so far.

We've given the function a name, and we've specified all the input data as arguments to the function. Since we know that we're trying to work out the monthly payments, we've created a variable to hold this information, and instructed the function to return it when it's finished.

The next thing we need to do is figure out how to combine all those arguments – all that input data – to get the output we're after – **monthlyPayments**. This is what might be called 'specific knowledge'. It's not really anything to do with programming, but it's some extra knowledge that we need to solve the problem. This is often a vital step in successfully completing a programming problem: identifying and finding the specific knowledge necessary to complete the problem. You can use any source you like, so long as you're confident that it's reliable (your program won't work properly if it's based on faulty assumptions). Knowing that we need some information about how to calculate mortgage repayments, a quick Google search turned up this handy and entirely relevant formula on Wikipedia:

```
c = rP / (1 - (1 + r)^N)
```

Here, **c** is the monthly payment, **r** the monthly interest as a decimal, **P** the amount borrowed and **N** the number of monthly payments. Notice how all the variables needed to calculate **c** in this formula are already provided to our function as arguments?

Putting it all together

Now we must convert this knowledge into our programming language of choice, filling out our function sketch.

One important thing to be wary of is that all of the inputs that we have are in the same units as the formula expects. Because they're not here, we've added a couple of lines to convert them.

Also, note that we have changed the argument names to match the formula. This isn't necessary, but consistency can help avoid mistakes.

```
import math

def monthlyPayments(P, r, N):
        r = r / 100.0 / 12.0
        N = N * 12
        monthlyPayments = (r * P) / (1 - math.pow (1 + r),
(N * 12 * -1))
        return monthlyPayments
```

And that's our function for calculating monthly payments finished. At this point, it's worth testing what we have put together so far.

It's obviously not the complete program, as it doesn't tell us anything about the amount of interest paid, but testing regularly will make spotting and fixing bugs much easier.

This is a vital step in the development process, and working like this is known as incremental development.

Before you can test code, you need some test cases to check it with. In this case, come up with a few values of **P**, **r** and **N**, and work them out using a calculator.

Be sure to take note of the expected result. Here's one that we came up with:

```
monthlyPayments (200000, 8, 30) = 1467.52
```

Now that we've established a test case, modify the code so that the function gets called with these arguments, and print the result.

If everything matches, that's great. If it doesn't, a good way of finding out where the problem lies is by adding more **print** statements to check that certain variables are what you'd expect them to be at that point in the code. For example:

```
r = r / 100.0 / 12.0
print r
```

would allow us to check that **r** was what we expected.

> "A good way of finding out where the problem lies is by adding more print statements"

With the first problem solved, we're ready to move on to the second: calculating interest paid. We'll leave you to do that, following the method laid out here, and instead look at how you can tie these two functions together.

› **Mike wants to live here, and he needs us to figure out whether he can afford it! (From Wikipedia's Innsbruck page.)**

The simplest way to do this is to create a third function that calls both the others and prints their output in a pretty format. This is what our final program looks like:

```
import math

def monthlyPayments(P, r, N):
        ...

def totalInterest(c, N, P):

def mortgageComp(P, r, N):
        print 'Monthly Payments:', monthlyPayments(P, r,
N)
        print 'Total Interest:', totalInterest(monthlyPayment
s(P, r, N), N, P)

mortgageComp(150000, 6, 25)
```

Designing programs

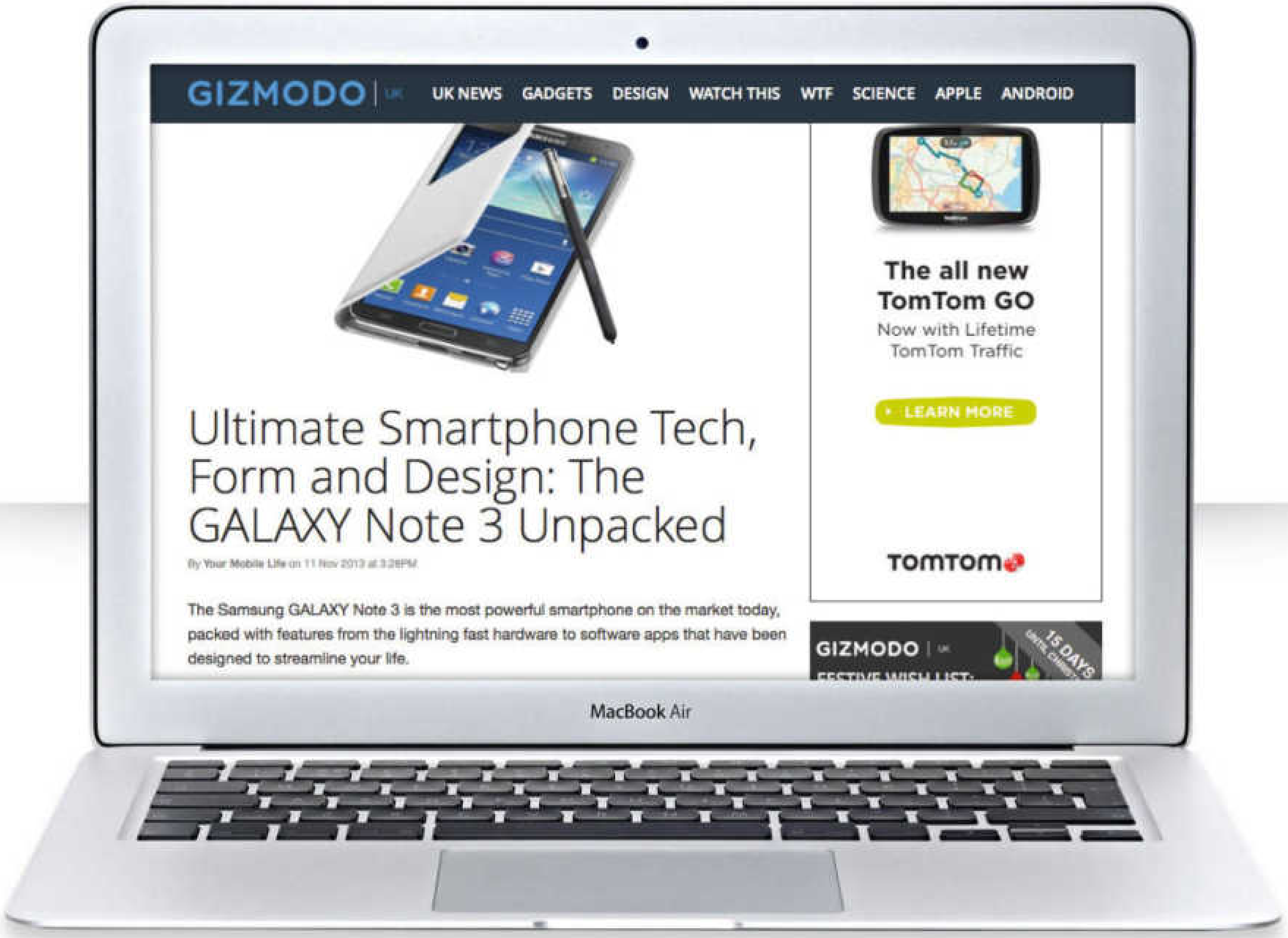
So, at the end of all this, what did our whole development process look like?

We began by critically reading the problem specification, and identifying the input and output data involved. By looking closely at this information, we managed to find a way to split the problem in two, making it easier and more manageable. We called this top-down development. We then developed solutions to each of these as separate functions, beginning with sketches and testing each as we progressed. This was called incremental development.

Also remember that, where we had to, we looked up specific knowledge which allowed us to solve the problem. Finally, with working solutions to all of the subproblems, we combined them to come up with a solution to the original, larger problem.

The steps are simple, but this is exactly what you would do to tackle almost any programming problem. For some, you might have to further divide the subproblems, but you would just keep going until you reached a level that was easily solvable and then work your way back, solving each level as and when you reached it. ■

T3
The Gadget Magazine
MORE GREAT CONTENT AT T3.COM
GET FIT FAST IN 2016 WITH THE VERY BEST TECH FOR RUNNING, CYCLING AND MORE...



Ultimate Smartphone Tech, Form and Design: The GALAXY Note 3 Unpacked

By Your Mobile Life on 11 Nov 2013 at 3:26PM

The Samsung GALAXY Note 3 is the most powerful smartphone on the market today, packed with features from the lightning fast hardware to software apps that have been designed to streamline your life.

Recursion: round and round we go

Jump down the ultimate rabbit hole of programming ideas, but be careful – you may well land on your own head.

Recursion is one of those concepts that sounds far more intimidating than it really is. Honestly. To make use of it, you don't need to grasp complicated mathematical ideas; or start thinking in four dimensions; or invade one level after another of someone's dreams to implant an idea into their subconscious; or even understand the meaning of recursive acronyms, such as GNU (GNU's not Unix).

For the most part, many uses of recursion are well documented and explored. You can then just copy and paste them into your code without necessarily understanding how they work, or why they do what they do. But if you do this, you're missing out, because recursion is cool. It's one of the best ways of letting the CPU do all the hard work while you write a few lines of code; and as a result, it can save you from bending your brain around what might be a more complex problem without recursion.

This is because recursion is simply a way of solving a more complex problem by repeating a simple function. The trick is that this function repeats from within itself. Rather than being run manually from another function, it calls another instance of itself from within its own code. It's the programming equivalent of pointing two mirrors together to create a seemingly infinite corridor, or connecting to the same virtual desktop from within a virtual desktop.

X factorial

In programming, one of the classic, and easiest, examples of recursion is a function to give a number's factorial. The factorial of a positive integer (we have to say this because the function would change otherwise) is the product achieved by multiplying the whole numbers it contains against one another. The factorial of 4, for example, is **4x3x2x1**, which equals 24.

If we wanted to approach this problem in the same way we've been doing with previous tutorials – a strategy known as iterative programming – we'd maybe write a function that multiplied two values within a **for** loop which was counting down through the factorial input value. In pseudo code, this might look something like the following:

```
function factorial(c)
        for i =c, i >= 1, i--
                total = total *i
        print total
```

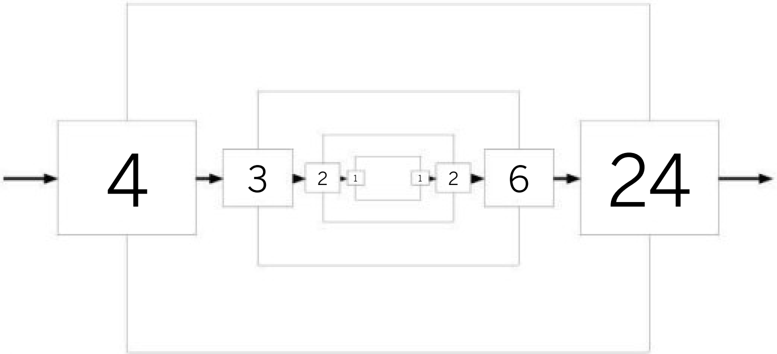
Pseudo code like this is used to explain a concept and to create a template to show the logic of a program, but the syntax isn't from any one specific language. It needs to be detailed and clear enough for the reader to understand the logic, without being weighed down by the peculiarities or eccentricities of one implementation. We've used a more C-like syntax for the **for** loop, for instance, because we think this is easier to read than the **range** keyword we'd need to use if we wrote the same code in Python. If you're working on a problem and don't know how to approach it, writing a pseudo code sketch of any potential solution is a great way of communicating your ideas or proving your concept works. In the above snippet, we create a **for** loop that uses a variable called **i** (that's a common name for variables within a **for** loop) to step through a range of values. We've used the C-syntax to say this value needs to hold a number more than (**>**) or equal (**=**) to the value **1**, and that it should be increased by one with every iteration of the loop. In other words, the value of **i** steps down from **c**, the input value, to the value **1**. Each iteration of the multiplication is controlled by the code, which is why this is an iterative approach.

You might already see from the previous piece of code that there's a slightly more efficient way of achieving the exact same result.

And that way uses recursion. Instead of controlling the function from a **for** loop, we're going to ask the function to control itself knowing that it has been called from within itself. That's confusing to read, but it's a concept much clearer in code. Here's the pseudo code for an updated factorial function that uses recursion instead:

```
function factorial(c)
        if c > 1
                return  c * factorial(c-1)
        else
                return 1
```

All that's happening here is that we've replaced the old factorial function with one that calls itself and expects a value in return. Don't forget that as well as passing values to a function, the function itself can return a value, usually using



› To calculate the factorial of a number, we can use a function that calls itself and returns the correct answer. That's recursion!

the **return** keyword, so that the code that called the function can use the value. You can see that while it's harder to understand the logic of how the final value is calculated, there's a lot less code than with the iterative approach, making this solution more efficient and less prone to errors. This is true of the majority of recursive solutions.

As long as the incoming value is greater than **1**, **factorial** calls itself with a new value that's one less than the one it was called with, and multiplies this by the value that's returned. If you call **factorial(4)**, for example, **factorial()** will be executed from within itself with the values **3**, **2** and then **1**. When it hits **1**, the return values start to trickle back because **factorial()** is no longer being called recursively. **1** is returned and multiplied by **2**, which itself returns **(2x1)** to the previous call, which returns **(3x2x1)** which, in turn, returns **(4x3x2x1)**. If you want to see this as Python code so you can try it out, type the following into the Python interpreter:

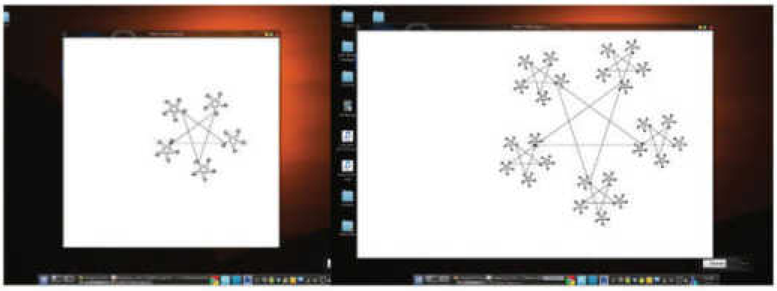
```
>>> def factorial(c):
...  if c > 1:
...   return c * factorial(c-1)
...  else:
...   return 1
...
>>> factorial(4)
24
>>> factorial(5)
120
>>>
```

The last two lines in that example are simply executing the **factorial()** function with the value we want to calculate. When the function finishes and returns the final calculation, the Python interpreter prints this out without us needing to do anything else, which is why you see the results under this line. Congratulations – you've solved the problem using recursion! You should now be able to see how recursion can be used in all sorts of places that require an indeterminate amount of repetition, such as many sorting algorithms, other mathematical solutions and filesystem searches. Even if you can't think of a solution yourself, if you have an idea that something can be approached using recursion, then a simple Google search will reveal whether it can and what the code, or pseudo code, should look like.

Generating fractals

But if you want to visualise recursion in action, which is one of the quickest ways to understand the concept, there's one particularly effective set of functions that can be used to illustrate both its advantages and its complexities, and these are fractals. Fractal algorithms use recursion to add infinite levels of detail to what's a much simpler algorithm. Some of the most famous were documented by the French mathematician Benoît B Mandelbrot in his book, *The Fractal Geometry of Nature*, but there are many easier algorithms that can be created with just a few lines of code, which is what we're going to do using Python.

Before we start, though, you will need to make sure you have two dependencies installed. The first is Python's Turtle graphics module, but it should be part of any default Python installation. We use this to draw and display lines with as little code as possible. The Turtle module itself needs you to install **tk-8.5** for its drawing routines, which means if your distribution doesn't add this automatically, you'll have to install it yourself. Using the Turtle graphics module means we can also see exactly what our script is doing as it's doing it, because the cursor moves slowly along the path as our fractal is being drawn.



› **We generate this fractal with just 16 lines of code. But its complexity is potentially infinite.**

The fractal we're going to create is called a star fractal. This is a five-pointed star, with another star drawn on to the end of each of its limbs, and a further one on each of those limbs, and so on, and so on. But to start with, we're going to create a function to draw a star, then expand this to draw another star at the correct point. Drawing with Turtle graphics is perfect for fractals because it doesn't draw lines with co-ordinates, but instead uses the relative position of a cursor and an angle to draw the path with the cursor as it moves forward.

```
import turtle as trtl
def fractal(length=100):
   if length < 10:
         return
   trtl.fd(length)
   trtl.left(144)
   trtl.fd(length)
   trtl.left(144)
   trtl.fd(length)
   trtl.left(144)
   trtl.fd(length)
   trtl.left(144)
   trtl.fd(length)
   trtl.left(144)
   return
```

If you run the above piece of code, either from the interpreter, by typing **fractal(200)**, or by adding that command to the last line in a script, you'll see the Turtle cursor first move forward before turning left 144 degrees and drawing another line. It does this four more times to complete the star. To turn this into a fractal, we want to execute the same function from within the function itself at the end of each limb, and this can be easily achieved by adding **fractal(length*0.3)** after every **trtl.fd(length)** line. This will call the fractal function again, only this time with a length value around a third smaller than the original. This star, too, will launch other iterations, but the reason why this doesn't go on for ever is because we first check whether the length of each line is going to be greater than 10. If it isn't, then we return from the function, and stars will stop being indefinitely drawn. As with the factorial function, this means that all the other part-completed stars can draw their respective limbs until the main star itself has finished and the fractal is complete. With just a few lines of code, you end up with what could be an infinitely complex image. And that's the never ending beauty of recursion. ■

Quick tip

If you want your Python application to pause before quitting, or doing anything else, add **from time import sleep** to the top of your script and use the **sleep** (seconds) command within your code. This is very useful if you want to see the final Turtle graphic rendering before the window closes.

Super sorting algorithms

Let's take our coding further and introduce you to the unruly world of algorithms, efficiency and sorted data.

Python is a great language. You're happily writing a little program, and you've got a huge list that you need to put in order. What do you do? Easy, just call the **sort()** method on the list, and you're away. It works amazingly quickly, and it doesn't even care how big the list is – it could be millions of elements long and still take virtually the same amount of time to return the sorted list.

This is really handy, but have you ever stopped to wonder how on earth it works? It's a fascinating question, and in the next two articles we're going to introduce you to some of the techniques that are used to solve this kind of problem. Along the way, you'll also get a gentle introduction to algorithms and how to think about their performance.

So, where do we start? Sorting a list with a computer, like many programming problems, seems incredibly abstract and difficult when you first begin. If you start off trying to think about how to sort a Python list with a million items in it, you won't get anywhere very quickly.

Sorting two cards

But what if you step back, away from the computer and the list of a million items? Imagine, for example, that you have two hearts cards from a deck. How would you put these two cards in numerical order?

Well, it's pretty simple: you'd look at them, and if the card on the left was worth more than the card on the right, you'd switch them around; if the card on the right was worth more, you'd leave them as they were.

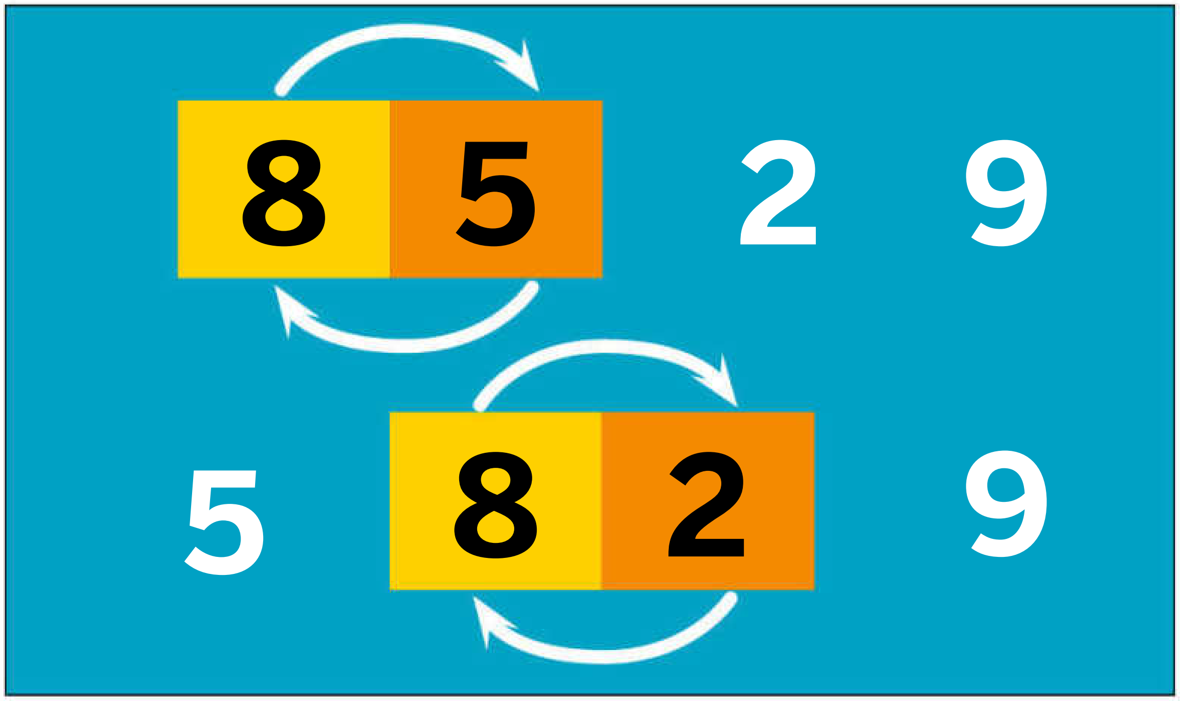
What would that look like in Python?

```
cards = [8,2]
if card[0] > card[1]:
   card[0] = card[1]
   card[1] = card[0]
print cards
```

That seems pretty straightforward. There's a list of cards, with the values **8** and **2**. We then check to see whether the first card is more valuable than the second, and if it is, copy each card to the correct location in the list.

Run that code and see what happens. You should find that you don't get **[2,8]**, but **[2,2]**. That's obviously not what we want, so what happened?

Because there's no operator in Python that enables us to switch the position of two elements, we did what seems most natural to us – we used the assignment operator to copy the value of the two cards. The problem is that after the first **copy** statement, we ended up with both cards having the same value. When we tried to do the second assignment, we just copied two identical cards.



Bubble sort works by comparing adjacent elements in the list. As it does so, the largest element 'bubbles' its way to the end of the list.

The way to get around this is to store a copy of the first card's value outside of the list, and before we do any copying. Take a look at this code:

```
cards = [8,2]
card0 = cards[0]
if card[0] > card[1]:
    card[0] = card[2]
    card[1] = card0
print cards
```

If you run that, you should get the correct answer. Because we stored a copy of the first card's value when we overwrote it with the first assignment statement, we could use the copy to set the second card to the correct value.

In the end, it's not quite how we would do things in real life, but it's pretty close.

Bubble sort

OK, so sorting a list two elements long isn't particularly impressive. But we can extend the same technique to sort a list that has an infinite number of elements – this is known as bubble sort.

The idea is that we loop over the entire list, comparing (and swapping, if necessary) each set of adjacent elements in turn. The element with the greatest value will always end up on the right-hand side of the comparison, so will always be compared in the next pair of elements. Eventually, it will end up at the very end of the list, where it belongs.

Notice that, if we were to loop over the list only once, we'd get only the largest element in the correct position. The way to get around this is to keep looping over the list; on each loop, we'll encounter another largest element that always ends up on the right-hand side of each comparison – that is, until it reaches the largest element from the previous iteration. At this point, it won't move any further, because it's now in the correct position.

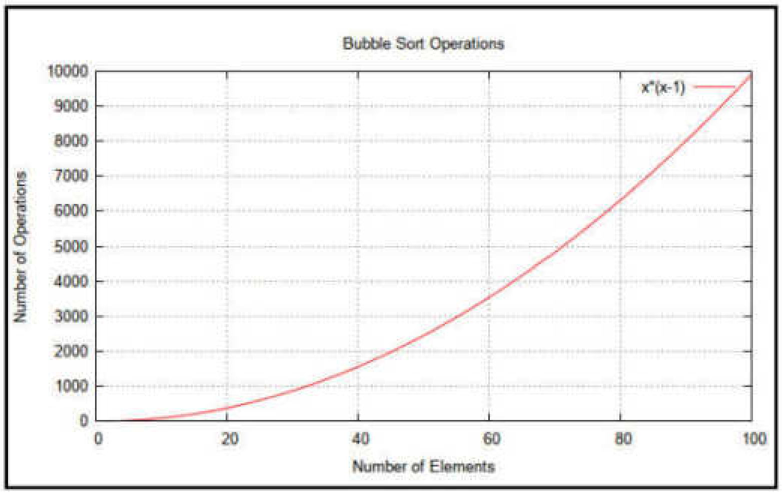
We'll know the list is sorted when we run a loop in which nothing gets swapped. This code shows how a bubble sort might be implemented in Python:

```
cards = [8, 3, 7, 4, 2, 1]
swapped = True
while swapped: #sort until swapped is False
    swapped = False #assume nothing is swapped
    for i in range(len(cards) - 1): #loop entire list
        cur = cards[i]
        j = i + 1
        if cards[i] > cards[j]:
            cards[i] = cards[j]
            cards[j] = cur
            swapped = True #reset swapped to True if anything
is swapped
print cards
```

Speed matters

It's pretty clever, right? You can now use this program to sort any size list and it will get the job done. However, if you try to use it on a large list, you'll find that it's painfully slow – even on a fast computer.

To understand why this is, let's think a little about how much work our bubble sort algorithm has to do. When we had just two cards, bubble sort had to do two comparisons – once to get the cards in the correct place, and then again to check that there's nothing else to sort. In total, it had to perform just two operations.



Bubble sort is an algorithm with O(n²) performance – that means it takes increasingly longer for every element added.

When we add a third card, assuming that they're in reverse order, bubble sort has to do the most work possible; it has do two comparisons on each loop, and three loops. In total – that's six operations.

What about if we add a fourth? It will take three comparisons on each loop, and four loops. That's 12 operations this time. Five cards? That's four comparisons and five loops: 20 operations.

There's a definite pattern here. The work done by bubble sort seems to conform to the formula:

num. operations = num. loops x num. comparisons

And we can see that the number of loops is always equal to the number of elements in the list, and the number of comparisons is always equal to one less than the number of loops. This means that, if we have **n** elements, the amount of work done by bubble sort is:

num. operations = n x (n - 1) = n(n - 1) = n² - n

Big O

What this means is that the amount of work bubble sort does increases in polynomial time for every item added to the list. That is to say the increase in the amount of work becomes more and more rapid for every element added to the list. This is why it's so slow for large lists – a 10-element list may take only 90 operations, but a 1,000-element list will take 999,000 operations, and it will simply get worse for every card from there on.

In the general case given above, the **n²** term is always going to be far larger than the **n** term, and hence will always have a much larger influence on how well the algorithm performs. Because of this, when discussing the performance of an algorithm, in computer science it's only ever the largest term that's considered.

The terminology used for this is 'big O' notation, and we say that bubble sort is a **O(n²)** algorithm. Bubble sort is actually considered to be one of the worst-performing sorting algorithms, and it's definitely not how Python does it. That said, don't discard it entirely. It's easy to implement, and works all right for small lists on fast computers. There may be times when you just want to get the job done, and bubble sort makes the most sense here.

If you fancy looking at sorting algorithms that are much faster, albeit harder to implement, jump to page 102, these are much closer to how Python does it. ■

Hidden secrets of numbers

Allow us to channel the living spirit of Dan Brown, and uncover the hidden meaning behind 1, 2 and -1b1010101.

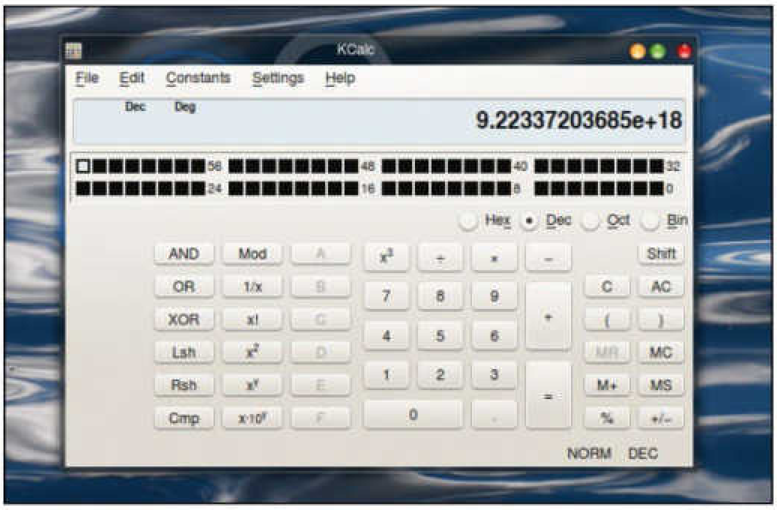
You wouldn't think that numbers would be a relevant topic in the 21st century. After all, if there's one thing computers can do well, it's count them. Billions of times a second. But for better or worse, having some knowledge of how computers deal with numbers, and the systems used to process them, will help you to write better programs. And it's not all for low-level stuff like assembler either. Number systems are just as important for high-level languages. If you write a PHP script for the web, for example, small changes in the way you process numbers within the script can have a huge impact on the performance of your servers. Your code might be run thousands of times a second, making any small discrepancy in how those numbers are calculated or stored have an impact on performance.

Numbers are important because they engage the CPU at its lowest level, right down to hardware and design. Typically, a single operation of the CPU can deal only with a value held within one of its internal registers. And that value can be as large only as the number of bits the CPU supports. These days, that means either 32 or 64 bits. But what's a bit? It's literally a switch that's turned on or off – a single bit of information, and this is normally represented by a **1** for **on** and a **0** for **off**. Computers use a numeral system called binary to store these on/off values within their registers and memory. The position of each bit within a binary value represents a number that's double the bit to the right of it, giving a sequence of bits the ability to represent every value between zero and double the value of the most significant bit minus 1 (or **2^8-1**). That's a difficult couple of sentences to digest, so here are some examples:

The positions within an 8-bit binary number have the following values: **1**, **2**, **4**, **8**, **16**, **32**, **64** and **128**. But rather than assign them in that order, the most significant bit – the one holding the largest value – is usually on the left. So, the binary number **10**, or **00000010**, is equivalent to **2**. If we move that bit to the left so that the **1** occupies the third slot along, the total value doubles to **4**, and so on. If we enable more than one bit at a time, then all the values are added together, so that **00000011** represents **3** (**1+2**) and **10000011** represents **131** (**128+2+1**). If we enable every bit in an 8-bit binary number, we get a maximum value of **255**.

> **"The position of each bit within a binary value represents a number that's double the bit to the right"**

› **If you want to see binary registers update in real time, KCalc has a handy display mode for up to 64 bits of data.**



2's compliment

Consider the following question: what does **10101010** represent in 2's compliment? You might think, after the last couple of paragraphs, that the binary value represented here is **170** (**2+8+32+128**). But you'd be wrong. The clue is the reference to 2's compliment. This is, in fact, a scheme for representing negative values in binary, and it changes the meaning of the bits and what they represent. It does this by reversing the configuration of bits, so that **1** becomes **0**, a process known as flipping, and adding **1** (there is a single exception to this).

So, the negative of **00000010** is first **11111101**, then with **1** added **11111110**, which is the 2's compliment representation of **-2**. Using this logic with the above question, we first subtract **1** from **10101010** to get **10101001** and flip the bits to **01010110**, which gives a decimal value of **86**, so the answer to the question is **-86**. The reason this method was chosen rather than swapping the most significant bit is because most binary arithmetic can still be performed on these negative values in exactly the same way they're performed on positive values. Adding **1** to **10101010** (**-86**), say, yields **10101011**, which is **-85** – the correct value.

While we're in the binary zone, it's worth mentioning a couple of other numeral systems sometimes used when programming. The first is octal, a base 8 system that uses the numerals 0-7 to hold 8 values. **7+1**, for example, is **10** in octal, just as **9+1=10** in decimal. It takes exactly three binary bits to represent a value in octal, which is why it's often used when you play with binary values. Similarly, hexadecimal, which is base 16, uses the characters 0-9 and a-f to represent decimal values 0-15, and that's the equivalent of four bits of data. So, the number **6c** in hex is **0110** (**6**) and **1100** (**12 = c**) in binary, which as an 8-bit value equates to **108**. Hex is a more readable equivalent to binary, which is why it's often used for memory and binary editors, and for storing raw binary data within your program.

You can avoid using these numeral systems in your code and your projects won't suffer. But understanding a little about what they mean and how they relate to one another can help you when looking at other projects, and how the underlying systems that take your code and make it run work. If you want to play with numeral systems, you can do so using Python's interpreter, and this is a great way of getting familiar with how the numbers work. From version 2.6 onwards, for example, you can enter a binary value using the **0b** prefix – that's a zero followed by the **b**. Typing a binary number will output the decimal value:

```
>>> 0b01010110
86
```

You can convert values to binary using the **bin** function:

```
>>> bin(86)
'0b1010110'
>>> bin(-85)
'-0b1010101'
```

As you can see with the last example, this also works with negative integers, returning a 2's compliment binary. If you want to input a 2's compliment binary, you need to prefix the data with **-0b**. You can do similar things with octal by replacing the **0b** with **0c** and with hexadecimal by using **0x**, and both of these numeral types have supporting functions for converting integers:

```
>>> hex(85)
'0x55'
>>> oct(85)
'0o125'
```

Data types

In many programming languages, you need to specify the type of a variable before you can use it. This can cause confusion in the beginner, because they wonder how they're supposed to know what they need to use before they write the code. But, in reality, you write the declaration for each variable as you come to use them.

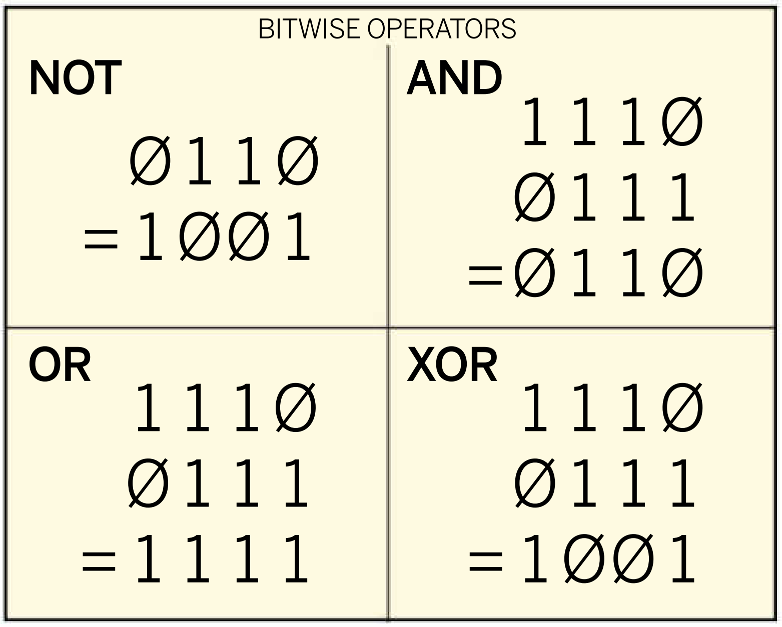
The compiler or interpreter needs to know about type because it wants to assign only enough memory and processing to handle that number, and no more. If you know your variable is never going to have a value greater than 255 (the value held by an 8-bit value), there's no need for the compiler to ensure more than this amount of storage is available. An 8-bit value has become known as a byte. Although the number of bits in a byte used to vary, it has settled on 8, because this is typically used to store a single character. A byte's worth of data is still used to store characters to this day, which you can view through ASCII codes with any reasonable binary editor.

But computer hardware has moved on, through the 16/32-bit era of the Atari ST and Amiga, to the pure 32-bit PCs of the last decade and to today, where most machines have a processor capable of handling 64-bit instructions. For that reason, like early bytes, there's no standard storage size for many data types. It always depends on your computer's hardware and its internal representation of a value as bits. The best example is an ordinary integer, which is most often defined as **int** in programming languages such as C and C++. The largest number an **int** can hold depends on the architecture of your CPU, and this limit is hidden within something called an **include** file for C and C++, and an equivalent for other languages. On most machines, you'll find that an unsigned **int** can hold a maximum value of **4294967295**, and if you paste this value into a binary calculator, you'll see this requires 32 bits of storage – probably the most common architecture in use. Even 64-bit machines can properly interpret 32-bit values.

This is why an **int** is usually the most generic variable type; it can store large numbers and can be addressed efficiently by most recent x86 CPUs. If you need larger, there's **long int** and if you need smaller, there's **short int**. You generally don't need to worry about any of this in languages such as Python, unless you want to, because the designers have decided to forgo the advantages of specifying the size of a variable for the simplicity it gives to programming. This is an example of something known as Duck typing, and it's another reason why Python is such a good beginner's language.

One set of types we've neglected to cover is those that include a floating point, such as **0.5** or **3.14159**. Rather than worry about the size of a float, it's the precision that's important. But unless you're dealing with small fractions, it's usually enough just to create a variable of type float to deal with these numbers.

The biggest problems when dealing with floating point numbers, and why programmers try to stick to integers, are introduced by rounding a value up or down from a more precise number, such as when rounding the value of pi to 3. While small rounding errors aren't likely to cause problems, they do when many of these are combined. This problem, and its dire consequences, was brilliantly illustrated by Richard Pryor's character in *Superman III* when he siphoned off the rounding errors in employees' payslips into his own account, and amassed a large fortune. ■



› **Binary and bitwise operators bring logic to your programs.**

Quick tip

Understanding how bits and bytes hold together is essential if you want to read or write to binary formats, or create a utility that communicates with external hardware.

Using loops and using loops

If we have explained how to use loops adequately, move to the next page. Otherwise, try reading this again.

After losing ourselves in the surprisingly complex domain of using numbers in our code, we're returning to a simpler concept. It's the idea of a loop, and how programmers use them to solve problems. A loop, in the programming sense, is a chunk of the same code that's designed by the programmer to be run over and over again. It's different from a function or a method, because while they're both designed to be re-run over and over again too, they're self-contained islands of logic.

With a method or a function, the programmer needs to know only what input is needed and what output to expect. Everything else should be handled by the function. A loop is a much more primitive construction, and without loops almost any programming task would become impossibly tedious. This is because they encapsulate what computers do well: repetition and iteration. Without loops, programmers are forced to write everything in longhand, and write code that can't accommodate unknown values or sizes. It's the difference between sowing a field by hand or by using Jethro Tull's horse-drawn seed drill.

Loops are so integral to code, we've already used several of them in our examples in previous tutorials in this bookazine. They're used as counters, as ways to step through data, for when you need to wait for a condition, and for filling arrays and files. But we've not spent any time discussing the various kinds of loops you can implement, or how you might attempt to solve a linear problem by adding a loop in your code. In many ways, they're related to the idea of recursion. But where a loop or function in recursion calls another instance of itself, usually to build a more complex solution, a loop on its own is used mostly to solve a simple calculation, or for waiting for a specific condition. Which is why, perhaps, the most common loop combines a conditional statement within a loop's structure – and that's the **for** keyword. The **for** loop is one of those peculiar parts of any language, because it requires a specific syntax that doesn't feel that logical. In the case of Python, a typical **for** loop looks like the following:

```
>>> for i in range (5):
...     print (i)
...
0
1
2
3
4
```

> **"Without loops, almost any programming task would become impossibly tedious"**

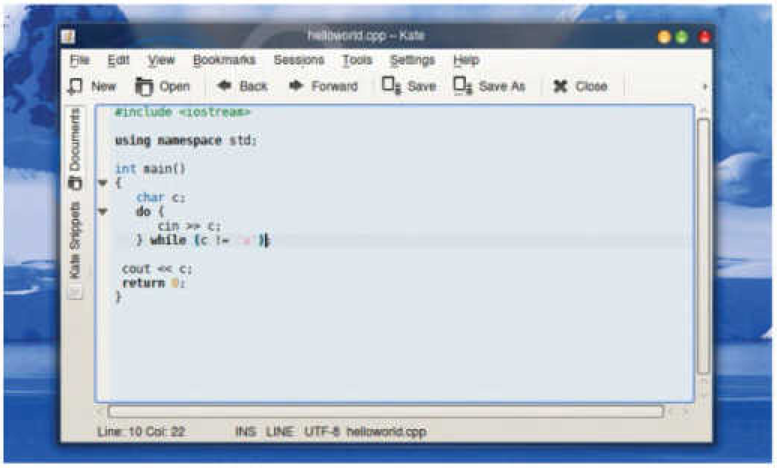
You can see in this snippet that the initial **for** statement requires a variable, which for some reason is nearly always called **i** in examples. Ours is no exception. This is followed in Python by the phrase **in range (5)**. **Range** is a special keyword in Python that we'll revisit in a couple of paragraphs. On the following line, we print the value of **i**. This line needs to be tabbed or spaced in because it's going to be executed as part of the for loop (as denoted by the **:**).

But without prior experience, you can't really guess at the output from the syntax of the **for** loop, and it's the same with other languages. This is because **for** is slightly different from most loops, because it includes the implicit definition of a variable. You just have to accept that whatever syntax they do use does the equivalent of defining a variable and creating an acceptable set of circumstances for the execution of your program to leave the loop. In our previous example, the variable **i** iterates between 0 and 4, leaving the loop when it gets to 5 (a total of five steps when you include zero). We print the value of **i** within the loop so you can see what's happening. If you write the same thing in C or C++, the syntax looks slightly different, but it gives you a better insight into what's happening:

```
for ( int i=0 ; i < 5 ; i++ ){
  cout << i << endl;
}
```

When compiled, linked and run (this is the big difference

› If you want to build your own C++ application, add the source code to a text file and build it with the command 'g++ helloworld.cpp -o helloworld'. Just run './helloworld' to see the results.


helloworld.cpp – Kate
File Edit View Bookmarks Sessions Tools Settings Help
New Open Back Forward Save Save As Close
Documents
Kate Snippets
Line: 10 Col: 22 INS LINE UTF-8 helloworld.cpp

```
#include <iostream>

using namespace std;

int main()
{
   char c;
   do {
      cin >> c;
   } while (c != [illegible]);

  cout << c;
  return 0;
}
```

between compiled languages such as C/C++ and interpreted languages such as Python and JavaScript), the output is identical to our Python example from earlier. In the C++ code, the variable **i** is declared as an integer (**int**) and incremented (**i++**) one value at a time for as long as it remains less than 5 (**<5**). This is exactly what the Python loop is doing, and the **cout** line is simply printing out the value of **i** during each iteration. You should be able to see from this that both the value of **i** and the value checked for by the condition can be changed, so you could get the value of **i** to count down instead by changing the loop to:

```
for ( int i=5 ; i > 0 ; i--){
  cout << i << endl;
}
```

To do the same in Python, we need to take a closer look at the **range** keyword used in the first example. **Range** is a function in Python that takes a start point, a stop point and a step size, much the same as the inputs for a C++ **for** loop. It's also much more versatile, as you don't need to restrict yourself to numbers – **range** can use indices of a sequence. If you put on one value in the range, as we did in the original loop, the function counts from zero up to the value **-1**. However, anything else will create a different effect. The advantage of farming it out to a function is that you're no longer restricted to using it in just the **for** loop, and you'll find many programmers use **range** for convenience throughout their code. If you wanted to count down in Python, for example, you could modify the **for** loop, as follows:

```
for i in range(5, 0, -1):
    print (i)
```

Infinite loop

If you don't need the iteration of a **for** loop, the best alternative is a **while** loop. This includes only the conditional check, and you're free to make that check positive in any way you choose. You could easily emulate the behaviour of a **for** loop, for example, by making the **while** loop wait for a variable to reach a certain value. Then, within your block of code, make sure your calculations attain that value at some point. In Python, we could recreate the previous **for** loop using **while**, like this:

```
>>> i = 5
>>> while i > 0:
...  i -= 1
...  print (i)
...
4
3
2
1
0
```

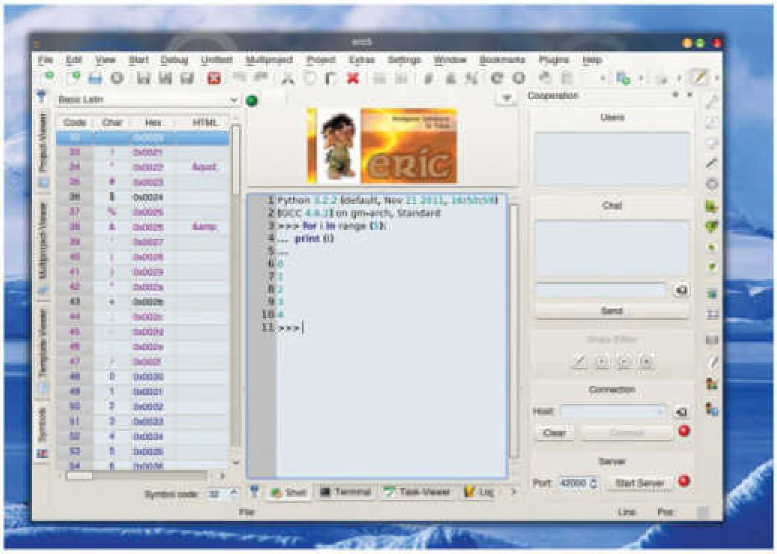
It's simpler this way, and by using **while** you can make the conditional checks for leaving a loop as complex as you need, rather than adhering to those required by **for**. Another trick in other languages, but not Python, is to place the conditional checking at the end of the block of code. This is normally called a **do...while** loop, because the **do** is used to start the section and the **while** is used to close the section.

Most importantly, the **while** statement at the end of the loop is checking the condition, such as 'do the washing up while there are still some dirty plates'. This can be helpful in certain circumstances, such as when you want the loop to take some input and then only test against that input at the end of the test. In C++, this function would look like this:

```
char c;
do {
  cin >> c;
} while (c != 'x');
```

This example is oversimplified, and isn't useful in itself, but it illustrates one example where **do...while** works well, and that's taking input. If you want to check the state of some input, whether it's from a file or from someone typing on the keyboard, you need to have first grabbed the input. You could do this in a normal **while** loop by asking for the input before you enter the loop, but it's slightly more efficient and elegant if you use a **do...while** loop to encapsulate the entire function. This is all the above loop is doing in C++. It grabs some input from the keyboard using **cin**, and waits for that input to consist only of the **x** character, after which it quits the loop. Another common use for **while** is to create what's known as an infinite loop – simply a loop that never satisfies its exit condition. This might sound like insanity if you ever want the processor to skip over the remainder of your code, but there are certain times, and languages, where the use of an infinite loop is the best way of building your application. One example is if you want to build a tool to monitor the state of a server or some other service.

You also find infinite loops watching hardware inputs or controlling the rendering within a game engine. Using an infinite loop means you can be sure your application is running, and hasn't exited under some other condition, and using **while** is probably the easiest way of creating one. **while (true) {}** is all that's needed. However, many languages also provide an escape from loops like this, and that's using something called the **break** command. **break** can be used to escape from an infinite loop, as well as leaving other kinds of loops early if you need to. You might want to escape from a **for** loop quickly, for example if you have detected that the user wants to quit the application. Using **break**, the execution of your code continues after the loop section, and you'd normally use this section of code to tidy up files, processes and memory before an exit. Of course, there's a good case for arguing against infinite loops, and building into your code the ability to escape on a specific condition, but this can take more effort and might not be as efficient, especially for smaller applications. ■



```
Python 3.2.2 (default, Nov 21 2011, 16:50:59)
[GCC 4.6.2] on gm-arch, Standard
>>> for i in range (5):
...   print (i)
...
0
1
2
3
4
>>>
```

Python is still a great language to learn with because it lets you experiment and see results as soon as you've typed your code.

Quick tip

Nearly all languages count zero as a value, such as the first position in an array. This seems illogical at first, because we're used to thinking of zero as nothing, but it's just something you have to get used to.

The magic of compilers

A compiler turns human-readable code into machine code, but there's a difference between compiled and interpreted languages.

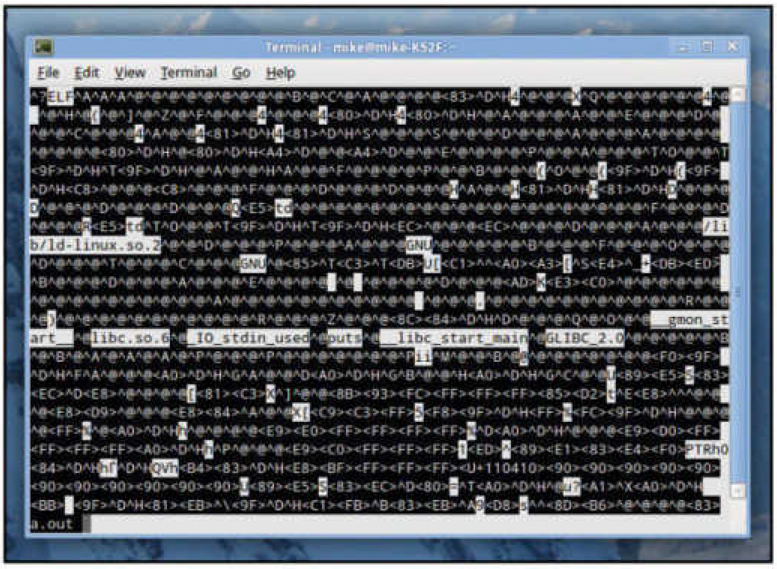
Computers aren't actually very smart. Sure, they can do a million things in the space of a second, and help us to transmit videos of cats around the internet, but they don't understand human languages. Ultimately, everything in a computer boils down to binary – ones and zeros – because that's all that a CPU understands. For us programmers, this makes things a bit complicated. We can't directly tell the CPU in English that we want to print a message on the screen, wait for a key to be pressed and so forth; we have to somehow tell the machine this information in binary. But then, imagine if your programs looked like this:

```
10010010 11100011
01011011 10001101
...
```

That would be a nightmare, wouldn't it? Finding mistakes would be nigh-on impossible, and if you gave your code to someone else for modification... Well, the entire concept of free/open source software falls apart. So, to fix this problem, we use compiled and interpreted languages (more on the latter later). In a compiled language, human-readable source code is processed, pulled apart, jumbled up and eventually converted into binary code that the CPU can understand. The program that does this is called – naturally enough – a compiler, and it's one of the most important components of an operating system.

> "In a compiled language, source code is processed, pulled apart and converted into binary code"

› **Try to read a binary executable file in a text editor, and you'll just see a load of messed-up characters like this.**



```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@<83>^D^H4^@^@^@X^Q^@^@^@^@^@^@4^@
^@^H^@[^@^]^@^Z^@^F^@^@^@4^@^@^@4<80>^D^H4<80>^D^H^@^A^@^@^@^@^A^@^@^@^E^@^@^@^D^@
^@^@^C^@^@^@4^A^@^@4<81>^D^H4<81>^D^H^S^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A^@^@^@
^@^@^@^@^@<80>^D^H^@<80>^D^H<A4>^D^@^@<A4>^D^@^@^@^E^@^@^@^@^P^@^@^A^@^@^@^T^O^@^@^T
<9F>^D^H^T<9F>^D^H^@^@^A^@^@^@^H^A^@^@^@^F^@^@^@^@^P^@^@^B^@^@^@(^O^@^@(<9F>^D^H(<9F>
^D^H<C8>^@^@^@<C8>^@^@^@^F^@^@^@^D^@^@^@^D^@^@^@H^A^@^@H<81>^D^HH<81>^D^HD^@^@^@
D^@^@^@^D^@^@^@^D^@^@^@Q<E5>td^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^F^@^@^@^D
^@^@^@R<E5>td^T^O^@^@^T<9F>^D^H^T<9F>^D^H<EC>^@^@^@<EC>^@^@^@^D^@^@^@^A^@^@^@/li
b/ld-linux.so.2^@^@^@^D^@^@^@^P^@^@^@^A^@^@^@GNU^@^@^@^@^@^B^@^@^@^F^@^@^@^O^@^@
^D^@^@^@^T^@^@^@^C^@^@^@GNU^@<85>^T<C3>^T<DB>UI<C1>^^<A0><A3>[^S<E4>^_ <DB><ED>
^B^@^@^@^D^@^@^@^A^@^@^@^E^@^@^@^@ ^@^@^@^@^D^@^@^@^@<AD>K<E3><C0>^@^@^@^@^@^@
^@^@^@^@^@^@^@^@^A^@^@^@^@^@^@^@^@^@ ^@^@^@^@.^@^@^@^@^@^@^@^@^@^@^R^@^@
^@)^@^@^@^@^@^@^@^@^@^@^@^R^@^@^@^Z^@^@^@<8C><84>^D^H^D^@^@^@^@^Q^@^O^@^@__gmon_st
art__^@libc.so.6^@_IO_stdin_used^@puts^@__libc_start_main^@GLIBC_2.0^@^@^@^@^@^B
^@^B^@^A^@^A^@^P^@^@^@^P^@^@^@^@^@^@^@^@ii^M^@^@^@^B^@2^@^@^@^@^@^@^@^@<F0><9F>
^D^H^F^A^@^@^@<A0>^D^H^G^A^@^@^@^D<A0>^D^H^G^B^@^@^@^H<A0>^D^H^G^C^@^@^@U<89><E5>S<83>
<EC>^D<E8>^@^@^@^@[<81><C3>X^]^@^@<8B><93><FC><FF><FF><FF><85><D2>t^E<E8>^^^@^@
^@<E8><D9>^@^@^@<E8><84>^A^@^@X[<C9><C3><FF>5<F8><9F>^D^H<FF>%<FC><9F>^D^H^@^@
^@<FF>%^@<A0>^D^Hh^@^@^@^@<E9><E0><FF><FF><FF><FF>%^D<A0>^D^Hh^@^@^@^@<E9><D0><FF>
<FF><FF><FF><A0>^D^Hh^P^@^@^@<E9><C0><FF><FF><FF>1<ED>^<89><E1><83><E4><F0>PTRh0
<84>^D^Hh[^D^HQVh<B4><83>^D^H<E8><BF><FF><FF><FF><U+110410><90><90><90><90><90>
<90><90><90><90><90><90>U<89><E5>S<83><EC>^D<80>=^T<A0>^D^H^@u?<A1>^X<A0>^D^H
<BB> <9F>^D^H<81><EB>^\<9F>^D^H<C1><FB>^B<83><EB>^A9<D8>s^^<8D><B6>^@^@^@^@<83>
a.out
```

Compilers exist for many different programming languages, but here we're going to focus on C, because it's the language used for many important projects, such as the Linux kernel. By far the most common compiler on Linux is **GCC**, from the GNU project, and that is what we are going to use here.

Dissecting a program

First up, consider this simple C program:

```
#include <stdio.h>
int main()
{
        puts("Hello, world!");
}
```

This simply prints the message "Hello world!" to the screen. If you've never seen C before, you might be a bit puzzled by some of the text, so here's a brief explanation: the **#include** line says that we want to use standard input and output routines (**stdio**). Then **main** says that this is the main part of our program (that is, where execution should begin), and the **puts** command means "put a string".

Enter this text into a file called **foo.c** and then run the following commands:

```
gcc foo.c
./a.out
```

The first command uses **GCC** to compile **foo.c** from the source code into a binary executable file called **a.out**, and the second line runs it. You can get more information about the resulting file by running **file a.out** – you'll see output similar to the following:

```
a.out: ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/
Linux 2.6.15, not stripped
```

If you try to examine the file yourself, using **less a.out** for instance, then you'll just see gobbledygook, as in the screenshot shown on the left. This is because we mere mortals can't read binary, and so **less** is trying to convert it into plain text characters.

This final, resulting file isn't intended for human consumption, but we can peek into some of the stages in the compilation process. First of all, enter this command:

```
gcc -E foo.c > foo.p
```

With the **-E** switch, we tell **GCC** that we don't want a binary executable file just yet; instead, we want only to see what happens after the first stage of processing. Open **foo.p** in a text editor and you'll see that it's much longer than the original program, because the contents of the **stdio.h** file have been included in it. It's only right at the bottom that you can see the **main** function and our **puts** instruction.

In larger, more complicated programs, the compiler does much more work in this 'pre-processor' stage. It expands macros, handles **#define** statements and places additional information into the file for debugging purposes. This file is then complete, with all the information that **GCC** needs to start converting it into machine code.

Assembly

But wait! There's an intermediate step between the C code and binary, and it's called assembly language. Assembly is a human-readable way of representing the instructions that the CPU executes. It's much lower-level than typical languages such as C and Python, where you can simply say "print a string for me". In assembly, you have to move memory around into the video buffer – for instance, telling the CPU exactly which instructions it should execute. So, now enter this:

```
gcc -S foo.c
```

This produces a file called **foo.s** that contains the assembly language version of our program. In other words, it's a list of CPU instructions written in a slightly more readable form. Most of it will look like complete gibberish if you have never touched assembly before, but these are the two most important lines:

```
movl    $.LC0, (%esp)
call    puts
```

Essentially, this places the location of our "Hello world!" text string into a register (a little like a variable), and then calls the **puts** (**put string**) function from the C library. These two lines are exact CPU instructions – as programmers, there is no way that we can break them down into smaller pieces. Given that modern CPUs execute billions of instructions per second, it can be quite humbling to see single instructions laid bare like this.

In the normal process of compilation (that is, without the **-E** or **-S** switches), **GCC** now runs an assembler to convert those human-readable instructions into their binary equivalents. It then adds extra information to the resulting file (**a.out**) to make it a valid Linux executable, and we have a program that's ready to run.

Jump back!

We can take the **a.out** file and go one step backwards in the process, to the assembly language stage, using the **objdump** command, like so:

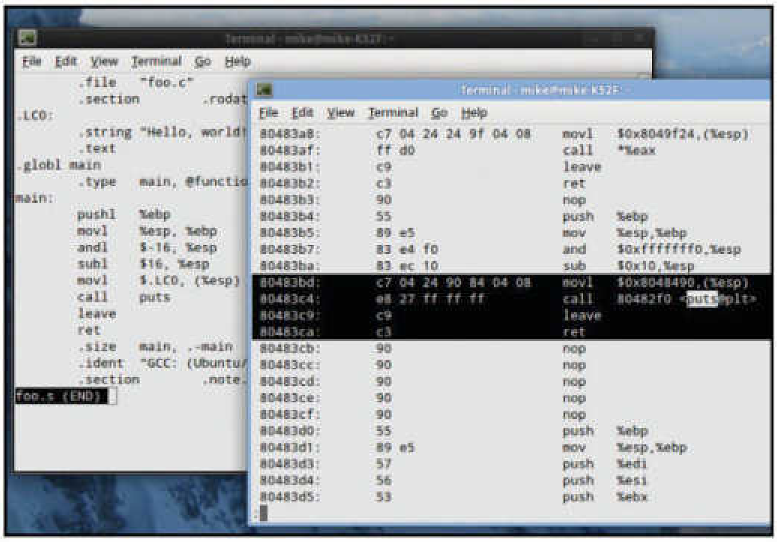
```
objdump -d a.out > list
```

Have a look at the **list** file now – it's a disassembly of the binary file, and therefore full of assembly language instructions. Most of them are unrelated to our program, and are used to tell the Linux program loader some useful things and set up the environment. But tucked away in there, you'll find these lines:

```
movl $0x8048490,(%esp)
call 80482f0 <puts@plt>
```

Those are the exact same instructions as the ones we saw a moment ago, expressed in a slightly different way. So, that's the process of how human-readable source code is converted into CPU-understandable instructions. The inner workings of a compiler are deep and complex, and many would argue that writing a fully-featured compiler for a complicated language such as C++ is an even harder job than writing an operating system. Think of all the work a compiler has to do in terms of managing variables, parsing complicated instructions, passing control around to different functions, and so on.

Don't think for a second that we've completely covered the compiler here. Shelves full of whopping great books have been written about the science of compilers – it's not a subject for the faint-hearted! But now you know the fundamentals, and you will certainly look at your code in a different way from this point on. ■


File Edit View Terminal Go Help
File Edit View Terminal Go Help

```
        .file   "foo.c"
        .section        .rodat
.LC0:
        .string "Hello, world!
        .text
.globl main
        .type   main, @functio
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    puts
        leave
        ret
        .size   main, .-main
        .ident  "GCC: (Ubuntu/
        .section        .note.
foo.s (END)
```

```
80483a8:       c7 04 24 24 9f 04 08    movl   $0x8049f24,(%esp)
80483af:       ff d0                   call   *%eax
80483b1:       c9                      leave
80483b2:       c3                      ret
80483b3:       90                      nop
80483b4:       55                      push   %ebp
80483b5:       89 e5                   mov    %esp,%ebp
80483b7:       83 e4 f0                and    $0xfffffff0,%esp
80483ba:       83 ec 10                sub    $0x10,%esp
80483bd:       c7 04 24 90 84 04 08    movl   $0x8048490,(%esp)
80483c4:       e8 27 ff ff ff          call   80482f0 <puts@plt>
80483c9:       c9                      leave
80483ca:       c3                      ret
80483cb:       90                      nop
80483cc:       90                      nop
80483cd:       90                      nop
80483ce:       90                      nop
80483cf:       90                      nop
80483d0:       55                      push   %ebp
80483d1:       89 e5                   mov    %esp,%ebp
80483d3:       57                      push   %edi
80483d4:       56                      push   %esi
80483d5:       53                      push   %ebx
```

› **On the left, the output from gcc -S, and on the right a disassembled binary. We've highlighted the areas showing identical instructions.**

What are interpreted languages?

There are two main types of programming language: compiled and interpreted. As we've seen in the former, a program called a compiler takes an entire source code file and converts it into CPU binary instructions in one fell swoop. In contrast, an interpreter reads individual lines from a source code file and acts on them one at a time, essentially interpreting each instruction at run time. For instance, take this BASIC program:

```
10 LET A = 1
20 PRINT "Hello world"
30 LET A = A + 1
40 IF A < 6 THEN GOTO 20
50 END
```

It's pretty clear what this does – it prints a message five times on the screen. Now, how would a compiler look at it? Because a compiler processes the source code file as a whole, it sees the bigger picture, knowing in advance that the loop is going to be executed five times. Consequently, the compiler can make various optimisations, such as replacing the whole bunch with five **PRINT** lines if that makes execution quicker. Similarly, the compiler can spot errors that might come up and warn the programmer about them.

An interpreter, on the other hand, is a much simpler program that simply steps through the program line-by-line. It doesn't know what's coming up, and just dutifully executes things as and when it sees them. This offers some advantages, in that you don't have to wait for a compilation process to finish before testing your work; and many interpreters allow you to pause them and make changes in the middle of execution.

Ultimately, most programmers prefer compiled languages over interpreted ones because of the performance benefits. Having your whole work condensed down into CPU-readable binary makes for better performance than a background program parsing each line step by step during execution.

Interpreted languages can, however, be useful where speed isn't crucially important, for example the Bash scripting language. Precision and individual execution rule here.

Avoiding common coding mistakes

Bug reports are useful, but you don't really want to cause too many. Here's what to avoid and how to avoid it.

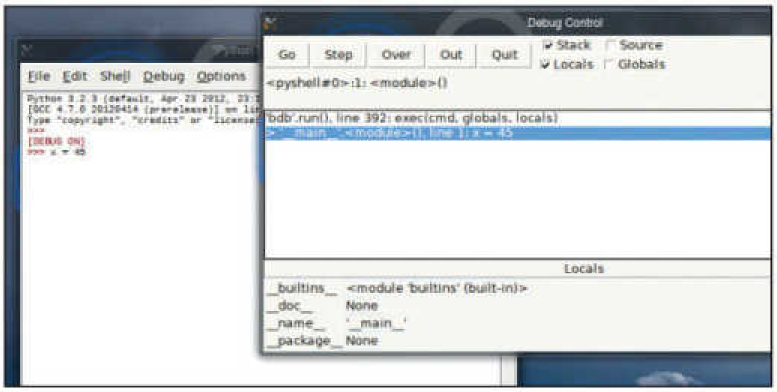
It doesn't matter how much care you put into writing your code. Even if you've had four cups of coffee and triple-check every line you write, sooner or later you are going to make a mistake. It might be as simple as a typo – a missing bracket or the wrong number, or it could be as complex as broken logic, memory problems or just inefficient code. Either way, the results will always be the same – at some point, your program won't do what you wanted it to. This might mean it crashes and dumps the user back to the command line. But it could also mean a subtle rounding error in your tax returns that prompts the Inland Revenue to send you a tax bill for millions of pounds, forcing you to sell your home and declare yourself bankrupt.

Finding the problems

How quickly your mistakes are detected and rectified is dependent on how complex the problem is, and your skills in the delicate art of troubleshooting. For instance, even though our examples of code from previous tutorials stretch to no more than 10 lines, you've probably needed to debug them as you've transferred them from these pages to the Python interpreter. When your applications grow more complex than just a few lines or functions, you can spend more time hunting down problems than you do coding. Which is why before you worry about debugging, you should follow a few simple rules while writing your code.

The first is that, while you can't always plan what you're going to write or how you're going to solve a specific problem, you should always go back and clean up whatever code you end up with. This is because it's likely you'll have used now-redundant variables and bolted on functionality into illogical places. Going back and cleaning up these areas makes the code easier to maintain and easier to understand. And making your project as easy to understand as possible becomes important as it starts to grow, and you seldom revisit these old bits of code.

› The IDLE Python IDE has a debug mode that can show how your variables change over time.



```
Python 3.2.3 (default, Apr 23 2012, 23:3
[GCC 4.7.0 20120414 (prerelease)] on lin
Type "copyright", "credits" or "license
>>>
[DEBUG ON]
>>> x = 45
```

```
<pyshell#0>:1: <module>()
```

```
'bdb'.run(), line 392: exec(cmd, globals, locals)
> '__main__'.<module>(), line 1: x = 45
```

```
__builtins__   <module 'builtins' (built-in)>
__doc__        None
__name__       '__main__'
__package__    None
```

Whenever you write a decent chunk of functionality, the second thing you should do is add a few comments to describe what it does and how it does it. Comments are simple text descriptions about what your code is doing, usually including any inputs and expected output. They're not interpreted by the language or the compiler – they don't affect how your code works, they are there purely to help other developers and users understand what a piece of code does. But, more importantly, they are there to remind you of what your own code does.

This might sound strange, but no matter how clear your insight might have been when you wrote it, give it a few days, weeks or months, and it may as well have been written by someone else for all the sense it now makes. And as a programmer, one of the most frustrating things you have to do is solve a difficult problem twice – once when you create the code, and again when you want to modify it but don't understand how it works. A line or two of simple description can save you days of trying to work out what a function calculates and how it works, or may even obviate the need for you to understand anything about what a piece of code does, as you need to know only the inputs and outputs.

The importance of documentation

This is exactly how external libraries and APIs work. When you install *Qt*, for instance, you're not expected to understand how a specific function works. You need only to study the documentation of the interface and how to use it within the context of your own code. Everything a programmer needs to know should be included in the documentation. If you want to use *Qt*'s excellent sorting algorithms, for example, you don't have to know how it manages to be so efficient, you need to know only what to send to the function and how to get the results back.

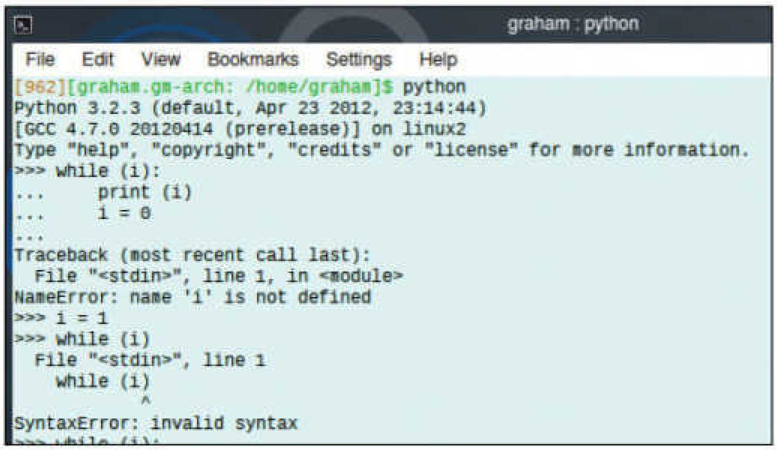
You should model your own comments on the same idea, both because it makes documentation easier, and because self-contained code functionality is easier to test and forget about. But we don't mean you need to write a book. Keep your words as brief as they need to be – sometimes that might mean a single line. How you add comments to code is dependent on the language you're using. In Python, for example, comments are usually demarcated by the **#** symbol in the first column of a line. Everything that comes after this symbol will be ignored by the interpreter, and if you're using an editor with syntax highlighting, the comment will also be coloured differently to make it more obvious. The more detail you put into a comment the better, but don't write a book. Adding comments to code can be tedious when you just want to get on with programming, so make them as brief as you can without stopping your flow. If necessary, you can go back and flesh out your thoughts when you don't feel like writing code (usually the day before a public release). When you start to code, you'll introduce many errors without realising it. To begin with, for example, you won't know what is and isn't a keyword – a word used by your chosen language to do something important. Each language is different, but Python's list of keywords is quite manageable, and includes common language words such as **and**, **if**, **else**, **import**, **class** and **break**, as well as less obvious words such as **yield**, **lambda**, **raise** and **assert**. This is why it's often a good idea to create your own variable names out of composite parts, rather than go with real words. If you're using an IDE, there's a good chance that its syntax highlighting will stop you from using a protected keyword.

Undeclared values

A related problem that doesn't affect Python is using undeclared values. This happens in C or C++, for instance, if you use a variable without first saying what type it's going to be, such as **int x** to declare **x** an integer. It's only after doing this you can use the variable in your own code. This is the big difference between compiled languages and interpreted ones. However, in both languages, you can't assume a default value for an uninitialised variable. Typing **print (x)** in Python, for instance, will result in an error, but not if you precede the line with **x = 1**. This is because the interpreter knows the type of a variable only after you've assigned it a value. C/C+ can be even more random, not necessarily generating an error, but the value held in an uninitialised variable is unpredictable until you've assigned it a value.

Typos are also common, especially in conditional statements, where they can go undetected because they are syntactically correct. Watch out for using a single equals sign to check for equality, for example – although Python is pretty good at catching these problems. Another type of problem Python is good at avoiding is inaccurate indenting. This is where conditions and functions use code hierarchy to split the code into parts. Python enforces this by breaking execution if you get it wrong, but other languages try to make sense of code hierarchy, and sometimes a misplaced bracket is all that's needed to create unpredictable results. However, this can make Python trickier to learn. Initially, if you don't know about its strict tabbed requirements, or that it needs a colon at the end of compound statement headers, the errors created don't make sense. You also need to be careful about case sensitivity, especially with keywords and your own variable names.

When you've got something that works, you need to test it – not just with the kind of values your application might expect, but with anything that can be input. Your code should fail gracefully, rather than randomly. And when you've got something ready to release, give it to other people to test. They'll have a different approach, and will be happier to break your code in ways you couldn't imagine. Only then will your code be ready for the wild frontier of the internet, and you'd better wear your flameproof jacket for that release. ■



```
[962][graham.gm-arch: /home/graham]$ python
Python 3.2.3 (default, Apr 23 2012, 23:14:44)
[GCC 4.7.0 20120414 (prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> while (i):
...     print (i)
...     i = 0
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'i' is not defined
>>> i = 1
>>> while (i)
  File "<stdin>", line 1
    while (i)
            ^
SyntaxError: invalid syntax
>>> while (i):
```

› You have to be careful in Python that the colons and indentation are in the correct place, or your script won't run. But this does stop a lot of runtime errors.

Comment syntax

Different languages mark comments differently, and there seems to be little consensus on what a comment should look like. However, there are a couple of rules. Most languages offer both inline and block comments, for example. Inline are usually for a single line, or a comment after a piece of code on the same line, and they're initiated by using a couple of characters before the comment. Block comments are used to wrap pieces of text (or code you don't want interpreted/compiled), and usually have different start and end characters.

Bash	**#** A hash is used for comments in many scripting languages. When **#** is followed by a **!** it becomes a shebang **#** and is used to tell the system which interpreter to use, for example: **#!/usr/bin/bash**
BASIC	**REM** For many of us, this is the first comment syntax we learn
C	**/*** This kind of comment in C can be used to make a block of text span many lines ***/**
C++	**//** Whereas this kind of comment is used after the **//** code or for just a single line
HTML	**<!--** Though not a programming language, we've included this because you're likely to have already seen the syntax, and therefore comments, in action **-->**
Java	**/**** Similar to C, because it can span lines, but with an extra ***** at the beginning ***/**
Perl	**= heading Overview** As well as the hash, in Perl you can also use something called Plain Old Documentation. It has a specific format, but it does force you to explain your code more thoroughly **=cut**
Python	**“’** As well as the hash, Python users can denote blocks of comments using a source code literal called a docstring, which is a convoluted way of saying ‘enclose your text in blocks of triple quotes’, like this **“’**

CODING

MADE SIMPLE

Further coding

Now you've got the basics down, it's time to advance your skills

Different types of Python data

Functions tell programs how to work, but it’s data that they operate on. Let’s go through the basics of data in Python.

In this article, we’ll be covering the basic data types in Python and the concepts that accompany them. In later articles, we’ll look at a few more advanced topics that build on what we do here: data abstraction, fancy structures such as trees, and more.

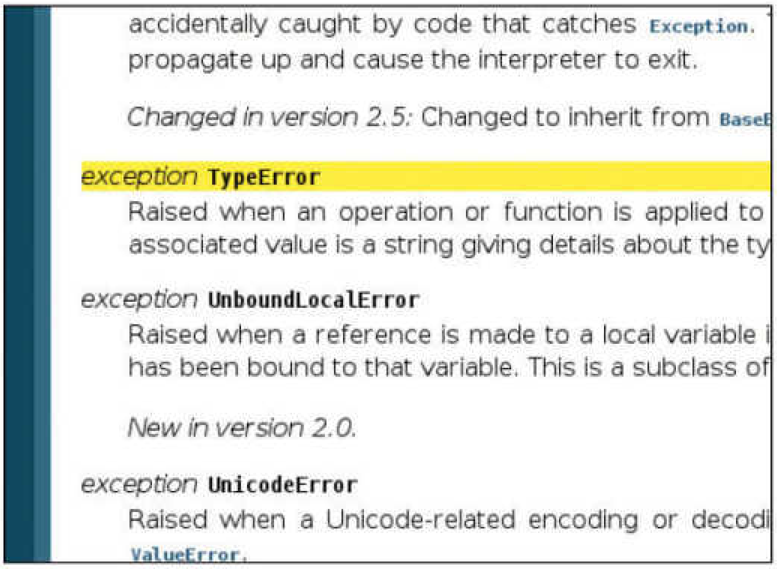
What is data?

In the world, and in the programs that we’ll write, there’s an amazing variety of different types of data. In a mortgage calculator, for example, the value of the mortgage, the interest rate and the term of the loan are all types of data; in a shopping list program, there are all the different types of food and the list that stores them – each of which has its own kind of data.

The computer’s world is a lot more limited. It doesn’t know the difference between all these data types, but that doesn’t stop it from working with them. The computer has a few basic ones it can work with, and that you have to use creatively to represent all the variety in the world.

We’ll begin by highlighting three data types: first, we have numbers. 10, 3 and 2580 are all examples of these. In particular, these are ints, or integers. Python knows about other types of numbers, too, including longs (long integers), floats (such as 10.35 or 0.8413) and complex (complex numbers). There are also strings, such as **‘Hello World’**, **‘Banana’** and **‘Pizza’**. These are identified as a sequence of characters enclosed within quotation marks. You can use either double or single quotes. Finally, there are lists, such as **[‘Bananas’, ‘Oranges’, ‘Fish’]**. In some ways, these are like a string, in that they are a sequence. What makes them different is that the elements that make up a list can be of any type. In this example, the elements are all strings, but you could create another list that mixes different types, such as **[‘Bananas’, 10, ‘a’]**. Lists are identified by the square brackets that enclose them, and each item or element within them is separated by a comma.

› While we’re looking only at basic data types, in real programs getting the wrong type can cause problems, in which case you’ll see a TypeError.



```
accidentally caught by code that catches Exception.
propagate up and cause the interpreter to exit.

Changed in version 2.5: Changed to inherit from BaseE

exception TypeError
   Raised when an operation or function is applied to
   associated value is a string giving details about the ty

exception UnboundLocalError
   Raised when a reference is made to a local variable i
   has been bound to that variable. This is a subclass of

   New in version 2.0.

exception UnicodeError
   Raised when a Unicode-related encoding or decodi
   ValueError.
```

Working with data

There are lots of things you can do with the different types of data in Python. For instance, you can add, subtract, divide and multiply two numbers and Python will return the result:

```
>>> 23 + 42
65
>>> 22 / 11
2
```

If you combine different types of numbers, such as an int and a float, the value returned by Python will be of whatever type retains the most detail – that is to say, if you add an int and a float, the returned value will be a float.

You can test this by using the **type()** function. It returns the type of whatever argument you pass to it.

```
>>> type(8)
<type ‘int’>
>>> type(23.01)
<type ‘float’>
>>> type(8 + 23.01)
<type ‘float’>
```

You can also use the same operations on strings and lists, but they have different effects. The **+** operator concatenates, that is combines together, two strings or two lists, while the ***** operator repeats the contents of the string or list.

```
>>> “Hello “ + “World”
“Hello World”
>>> [“Apples”] * 2
[“Apples”, “Apples”]
```

Strings and lists also have their own special set of operations, including slices. These enable you to select a particular part of the sequence by its numerical index, which begins from 0.

```
>>> word = “Hello”
>>> word[0]
‘H’
>>> word[3]
‘l’
>>> list = [‘banana’, ‘cake’, ‘tiffin’]
>>> list[2]
‘tiffin’
```

Indexes work in reverse, too. If you want to reference the last element of a list or the last character in a string, you can use the same notation with a **-1** as the index. **-2** will reference the second-to-last character, **-3** the third, and so on. Note that when working backwards, the indexes don't start at **0**.

Methods

Lists and strings also have a range of other special operations, each unique to that particular type. These are known as methods. They're similar to functions such as **type()** in that they perform a procedure. What makes them different is that they're associated with a particular piece of data, and hence have a different syntax for execution.

For example, among the list type's methods are **append** and **insert**.

```
>>> list.append('chicken')
>>> list
['banana', 'cake', 'tiffin', 'chicken']
>>> list.insert(1, 'pasta')
>>> list
['banana', 'pasta', 'cake', 'tiffin', 'chicken']
```

As you can see, a method is invoked by placing a period between the piece of data that you're applying the method to and the name of the method. Then you pass any arguments between round brackets, just as you would with a normal function. It works the same with strings and any other data object, too:

```
>>> word = "HELLO"
>>> word.lower()
'hello'
```

There are lots of different methods that can be applied to lists and strings, and to tuples and dictionaries (which we're about to look at). To see the order of the arguments and the full range of methods available, you'll need to consult the Python documentation.

Variables

In the previous examples, we used the idea of variables to make it easier to work with our data. Variables are a way to name different values – different pieces of data. They make it easy to manage all the bits of data you're working with, and greatly reduce the complexity of development (when you use sensible names).

As we saw above, in Python you create a new variable with an assignment statement. First comes the name of the variable, then a single equals sign, followed by the piece of data that you want to assign to that variable.

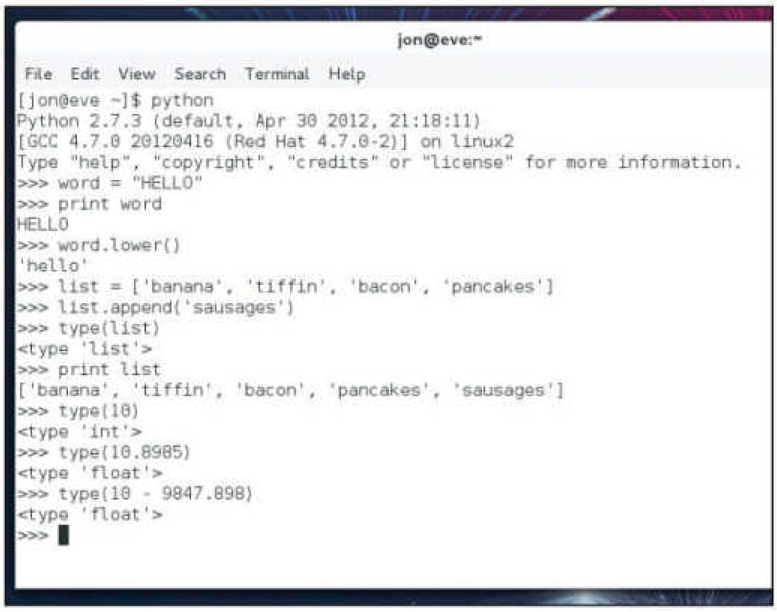
From that point on, whenever you use the name assigned to the variable, you are referring to the data that you assigned to it. In the examples, we saw this in action when we referenced the second character in a string or the third element in a list by appending index notation to the variable name. You can also see this in action if you apply the **type()** function to a variable name:

```
>>> type(word)
<type 'str'>
>>> type(list)
<type 'list'>
```

Other data types

There are two other common types of data that are used by Python: tuples and dictionaries.

Tuples are very similar to lists – they're a sequence data type, and they can contain elements of mixed types. The big difference is that tuples are immutable – that is to say, once you create a tuple you cannot change it – and that tuples are identified by round brackets, as opposed to square brackets: **('bananas', 'tiffin', 'cereal')**. Dictionaries are similar to a list or a tuple in that they contain a collection of related items. They differ in that the elements aren't indexed by numbers, but by 'keys' and are created with curly brackets: **{}**. It's quite like an English language dictionary. The key is the word that you're looking up, and the value is the definition of the word.



```
[jon@eve ~]$ python
Python 2.7.3 (default, Apr 30 2012, 21:18:11)
[GCC 4.7.0 20120416 (Red Hat 4.7.0-2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> word = "HELLO"
>>> print word
HELLO
>>> word.lower()
'hello'
>>> list = ['banana', 'tiffin', 'bacon', 'pancakes']
>>> list.append('sausages')
>>> type(list)
<type 'list'>
>>> print list
['banana', 'tiffin', 'bacon', 'pancakes', 'sausages']
>>> type(10)
<type 'int'>
>>> type(10.8985)
<type 'float'>
>>> type(10 - 9847.898)
<type 'float'>
>>> 
```

› The Python interpreter is a great place to experiment with Python code and see how different data types work together.

With Python dictionaries, however, you can use any immutable data type as the key (strings are immutable, too), so long as it's unique within that dictionary. If you try to use an already existing key, its previous association is forgotten completely and that data lost for ever.

```
>>> english = {'free': 'as in beer', 'linux': 'operating system'}
>>> english['free']
'as in beer'
>>> english['free'] = 'as in liberty'
>>> english['free']
'as in liberty'
```

Looping sequences

One common operation that you may want to perform on any of the sequence types is looping over their contents to apply an operation to every element contained within. Consider this small Python program:

```
list = ['banana', 'tiffin', 'burrito']
for item in list:
    print item
```

First, we created the list as we would normally, then we used the **for... in...** construct to perform the **print** function on each item in the list. The second word in that construct doesn't have to be **item**, that's just a variable name that gets assigned temporarily to each element contained within the sequence specified at the end. We could just as well have written **for letter in word** and it would have worked just as well.

That's all we have time to cover in this article, but with the basic data types covered, we'll be ready to look at how you can put this knowledge to use when modelling real-world problems in later articles.

In the meantime, read the Python documentation to become familiar with some of the other methods that it provides for the data types we've looked at before. You'll find lots of useful tools, such as **sort** and **reverse**! ■

More Python data types

Learn how different types of data come together to solve a real problem as we write some code that counts words.

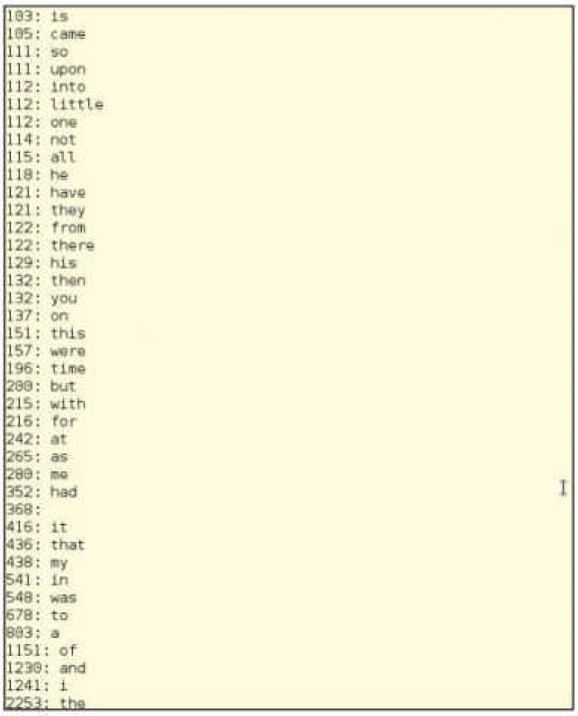
In the previous tutorial, we introduced Python's most common data types: numbers (ints and floats), strings, lists, tuples and dictionaries. We demonstrated how they work with different operators, and explained a few of their most useful methods. We didn't, however, give much insight into how they might be used in real situations. In this article, we're going to fix that.

We're going to write a program that counts the number of times each unique word occurs in a text file. Punctuation marks will be excluded, and if the same word occurs but in different cases (for example, **the** and **The**), they will be taken to represent a single word. Finally, the program will print the results to the screen. It should look like this:

```
the: 123
you: 10
a: 600
...
```

As an example, we'll be using *The Time Machine*, by HG Wells, which you can download from Project Gutenberg, saving it in the same folder as your Python file under the name **timemachine.txt**.

As the program description suggests, the first thing we'll need to do is make the text accessible from inside our Python program. This is done with the **open()** function:



```
103: is
105: came
111: so
111: upon
112: into
112: little
112: one
114: not
115: all
118: he
121: have
121: they
122: from
122: there
129: his
132: then
132: you
137: on
151: this
157: were
196: time
200: but
215: with
216: for
242: at
265: as
280: me
352: had
368:
416: it
436: that
438: my
541: in
548: was
678: to
803: a
1151: of
1230: and
1241: i
2253: the
```

› **Hardly surprisingly, our counting program, after being sorted, finds 'the' to be the most common word in *The Time Machine*, by HG Wells.**

```
tm = open('timemachine.txt', 'r')
```

In this example, **open()** is passed two variables. The first is the name of the file to open; if it were in a different directory from the Python script, the entire path would have to be given. The second argument specifies which mode the file should be opened in: **r** stands for **read**, but you can also use **w** for write or **rw** for **read-write**.

Notice we've also assigned the file to a variable, **tm**, so we can refer to it later in the program.

With a reference to the file created, we also need a way to access its contents. There are several ways to do this, but today we'll be using a **for... in...** loop. To see how this works, try opening **timemachine.txt** in the interactive interpreter and then typing:

```
>>> for line in tm:
      print line
...
```

The result should be every line of the file printed to the screen. By putting this code in to a **.py** file, say **cw.py**, we've got the start of our Python program.

Cleaning up

The program description also specified that we should exclude punctuation marks, consider the same word but in different cases as one word, and that we're counting individual words, not lines. As it stands, we have been able to read only entire lines as strings, however, with punctuation, strange whitespace characters (such as **\r\n**) and different cases intact.

Looking at the Python string documentation (**http://docs.python.org/library**), we can see that there are four methods that can help us convert line strings into a format closer to that specified by the description: **strip()**, **translate()**, **lower()** and **split()**.

Each of these are methods, and as such they're functions that are applied to particular strings using the dot notation. For example, **strip()**, which removes specified characters from the beginning and end of a string, is used like this:

```
>>> line.strip()
```

When passed with no arguments, it removes all whitespace characters, which is one of the jobs we needed to get done.

The function **translate()** is a method that can be used for removing a set of characters, such as all punctuation marks, from a string. To use it in this capacity, it needs to be passed two arguments, the first being **None** and the second being the list of characters to be deleted.

```
>>> line.translate(None, '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~')
```

lower() speaks for itself, really – it converts every character in a string to lower-case. **split()** splits distinct elements inside a string in to separate strings, returning them as a list. By passing an argument to **split()**, it's possible to specify which character identifies the end of one element and the start of another.

```
>>> line.split(' ')
```

In this example, we've passed a single space as the character to split the string around. With all punctuation removed, this will create a list, with each word in the string stored as a separate element.

Put all of this in the Python file we started working on earlier, inside the **for** loop, and we've made considerable progress. It should now look like this:

```
tm = open('timemachine.txt', 'r')
for line in tm:
    line = line.strip()
    line = line.translate(None, '!"#$%&\'()*+,-./:;<=>?@
    [\\]^_`{|}~')
    line = line.lower()
    list = line.split(' ')
```

Because all of the string methods return a new, modified string, rather than operating on the existing string, we've re-assigned the line variable in each line to store the work of the previous step.

Uniqueness

Phew, look at all that work we've just done with data! By using the string methods, we've been able to remove all the bits of data that we weren't interested in. We've also split one large string, representing a line, into smaller chunks by converting it to a list, and in the process got to the exact, abstract concept we're most interested in: words.

Our stunning progress aside, there is still some work to be done. We now need a way to identify which words are unique – and not just in this line, but in every line contained within the entire file.

The first thing that should pop into your head when thinking about uniqueness is of a dictionary, the key-value store we saw in the previous article. It doesn't allow duplicate keys, so by entering each word as a key within a dictionary, we're guaranteed there won't be any duplicates.

What's more, we can use the value to store the number of times each word has occurred, incrementing it as the program comes across new instances of each key.

Start by creating the dictionary, and ensuring that it persists for the entire file – not just a single line – by placing this line before the start of the **for** loop:

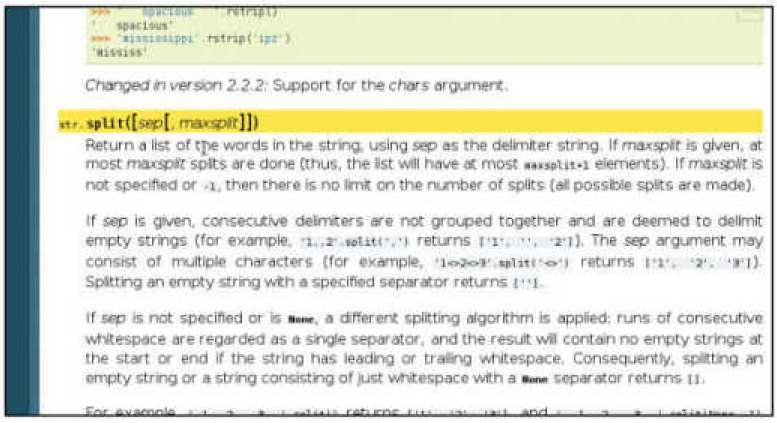
```
dict = {}
```

This creates an empty dictionary, which is ready to receive our words.

Next, we need to think about a way to get each word into the dictionary. As we saw last time, ordinarily a simple assignment statement would be enough to add a new word to the dictionary. We could then iterate over the list we created above (using another **for** loop), adding each entry to the dictionary with a value of **1** (to represent that it has occurred once in the file).

```
for word in list:
    dict[word] = 1
```

But remember, if the key already exists, the old value is overwritten and the **count** will be reset. To get around this, we can place an **if-else** clause inside the loop:



```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

Changed in version 2.2.2: Support for the *chars* argument.

str.split([*sep*[, *maxsplit*]])

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters (for example, `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). Splitting an empty string with a specified separator returns `['']`.

If *sep* is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns `[]`.

For example,

› Python's Standard Library reference, http://docs.python.org/library, is an invaluable source for discovering what methods are available and how to use them.

```
if word in dict:
    count = dict[word]
    count += 1
    dict[word] = count
else:
    dict[word] = 1
```

This is a little bit confusing because **dict[word]** is being used in two different ways. In the second line, it returns the value and assigns it to the variable count, while in the fourth and seventh lines, **count** and **1** are assigned to that key's value, respectively.

Notice, too, that if a word is already in the dictionary, we increment the **count** by 1, representing another occurrence.

Putting it together

Another data type wrestled with, another step closer to our goal. At this point, all that's left to do is insert some code to print the dictionary and put it all together and run the program. The **print** section should look like this and be at the very end of the file, outside of the line-looping code.

```
for word,count in dict.iteritems():
    print word + ": " + str(count)
```

This **for** loop looks different to what you've seen before. By using the **iteritems** method of the dictionary, we can access both the key (**word**) and value (**count**) in a single loop. What's more, we've had to use the **str()** function to convert **count**, an integer, into a string, as the **+** operator can't concatenate an integer and a string.

Try running it, and you should see your terminal screen filled with lines like:

```
...
other: 20
sick: 2
ventilating: 2
...
```

Data everywhere!

That's all we planned to achieve in this particular tutorial and it's actually turn out to be quite a lot. As well as having had a chance to see how several different types of data and their methods can be applied to solve a real problem, we hope you've noticed how important it is to select the appropriate type for representing different abstract concepts.

For example, we started off with a single string representing an entire line, and we eventually split this into a list of individual strings representing single words. This made sense until we wanted to consider unique instances, at which point we put everything in to a dictionary.

As a further programming exercise, why not look into sorting the resulting dictionary in order to see which words occur most frequently? You might also want to consider writing the result to a file, one entry on a line, to save the fruits of your labour. ■

Reliability by abstraction

Think your code is solid? Perhaps it's a bit too lumpy. Creating abstractions can make code much easier to maintain.

In the previous few tutorials, we have been looking at data. First, we introduced some of Python's core data types, and then we demonstrated how they can be put to use when solving a real problem. The next data-related topic we want to consider is abstraction, but before we get on to that, we're first going to look at abstraction in general and as it applies to procedures. So, this time we'll take a brief hiatus from data, before returning to it in a later article.

Square roots

To get our heads around the concept of abstraction, let's start by thinking about square roots and different techniques for finding them. One of these was discovered by Newton, and is thus known as Newton's method.

It says that when trying to find the square root of a number (x), we should start with a guess (y) of its square root; we can then improve upon that result by averaging our guess (y) with the result of dividing the number (x) by our guess (y). As we repeat this procedure, we get closer and closer to the square root. In most attempts, we'll never reach a definite result, we'll only make our guess more and more accurate. Eventually, we'll reach a level of accuracy that is good enough for our needs and then give up. Just to be clear about what is involved, take a look at the table below for how you would apply this method to find the square root of 2 (for example, x).

It's a lot of work just to find the square root of a number. Imagine if when you were in school, every time you had to find a square root you had to do all these steps manually. Solving problems involving Pythagoras' theorem, for instance, would be much more unwieldy.

> "This is a very powerful technique that makes programming easier"

Luckily, assuming you were allowed to use calculators when you were at school, there's another, much easier method to find square roots. Calculators come with a button marked with the square root symbol, and all you have to do is press this button once – much easier. This second approach is what's known as an abstraction. When working on problems, such as those involving Pythagoras' theorem, we don't care about how to calculate the square root, only that we can do it and get the correct result. We can treat the square root button on our calculator as a black box – we never look inside it, we don't know how it does what it does, all that matters is that we know how to use it and that it gives the correct result.

This a very powerful technique, which can make programming a lot easier, because it helps us to manage complexity. To demonstrate how abstraction can help us, consider the Python code below for finding the longest side of a right-angled triangle:

```
import math
def pythag(a, b):
    a2b2 = (a * a) + (b * b)
    guess = 1.0
    while (math.fabs((guess * guess) - a2b2) > 0.01):
        guess = (((a2b2 / guess) + guess) / 2)
    return guess
```

The first thing to note is that it's not in the least bit readable. Sure, with a piece of code this short, you can read through it reasonably quickly and figure out what's going on, but at a glance it's not obvious, and if it were longer and written like this, you'd have a terrible time figuring out what on earth it was doing. What's more, it would be very difficult to test the different parts of this code as you go along (aka incremental development, vital for building robust software).

For instance, how would you break out the code for testing whether or not a guess is close enough to the actual result (and can you even identify it?), or the code for improving a guess, to check that it works? What if this function didn't return the expected results – how would you start testing all the parts to find where the error was?

Finally, there's useful code in here that could be reused in other functions, such as that for squaring a number, for taking an average of two numbers, and even for finding the square root of a number, but none of it is reusable because of the way it's written. You could type it all out again, or copy and paste it, but the more typing you have to do, the more obscure code you have to copy and paste, and the more likely mistakes are to make it in to your programming.

Let's try writing that code again, this time coming up with some abstractions to fix the problems listed above. We haven't listed the contents of each new function we've

Find a square root

Guess (y)	Division (x/y)	Average (((x/y) + y)/2)
1	2/1 = 2	(2 + 1)/2 = 1.5
1.5	2/1.5 = 1.33	(1.33 + 1.5)/2 = 1.4167
1.4167	2/1.4167 = 1.4118	(1.4118 + 1.4167)/2 = 1.4142

created, leaving them for you to fill in.

```
import math:
def square(x):
    ...
def closeEnough(x, guess):
    ...
def improveGuess(x, guess):
    ...
def sqrt(x, guess):
    ...
def pythag(a, b):
    a2b2 = square(a) + square(b)
    return sqrt(a2b2)
```

Here, we've split the code in to several smaller functions, each of which fulfils a particular role. This has many benefits.

For starters, how much easier is the **pythag()** function to read? In the first line, you can see clearly that **a2b2** is the result of squaring two numbers, and everything below that has been consolidated in to a single function call, the purpose of which is also obvious.

What's more, because each part of the code has been split into a different function, we can easily test it. For example, testing whether **improveGuess()** was doing the right thing would be very easy – come up with a few values for x and guess, do the improvement by hand, and then compare your results with those returned by the function.

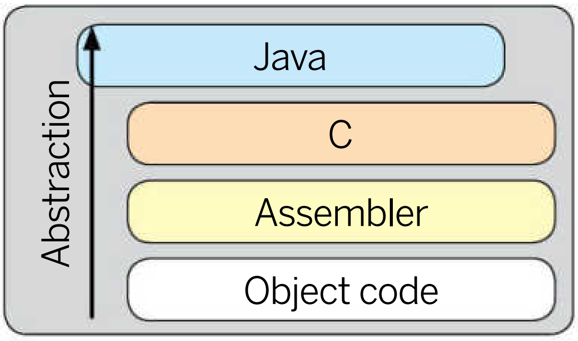
If **pythag()** itself was found not to return the correct result, we could quickly test all these auxiliary functions to narrow down where the bug was.

And, of course, we can easily reuse any of these new functions. If you were finding the square root of a number in a different function, for instance, you could just call the **sqrt()** function – six characters instead of four lines means there's far less opportunity to make mistakes.

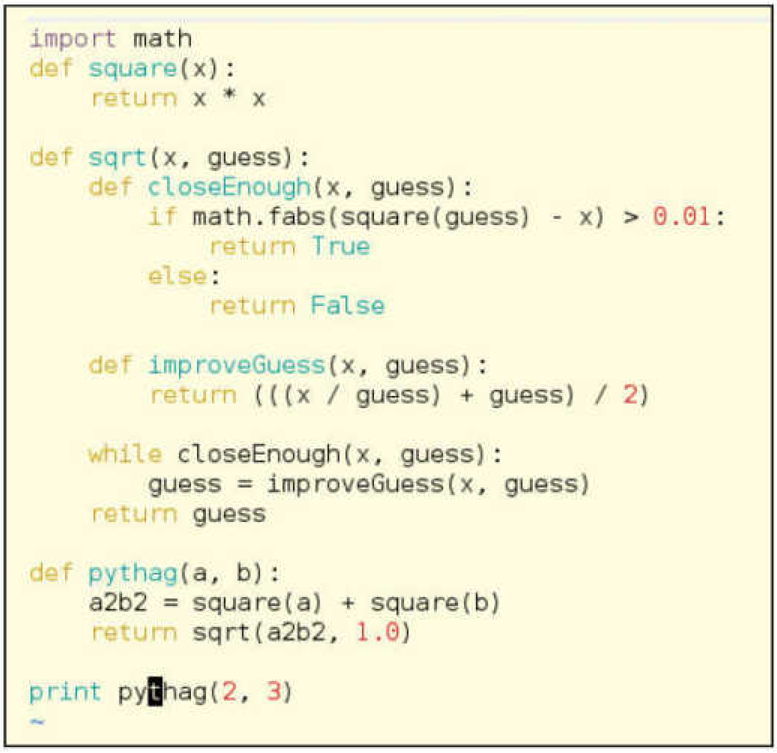
One final point: because our **sqrt** code is now abstracted, we could change the implementation completely, but so long as we kept the function call and arguments the same, all code that relies on it would continue to work properly.

> **"This code can be improved by taking advantage of scope"**

This means that if you come across a much more efficient way of calculating square roots, you're not stuck with working through thousands of lines of code, manually changing every section that finds a square root; you do it once, and it's done everywhere. This code can be improved still further by taking advantage of scope. **closeEnough()** and **improveGuess()** are particular to the **sqrt()** function – that is to say, other functions are unlikely to rely on their services. To help keep our code clean, and make the relationship between these functions and **sqrt()** clear, we can place their definitions inside the definition of **sqrt()**:



› There are layers of abstraction underneath everything you do on a PC – you just don't often think of them.



```
import math
def square(x):
    return x * x

def sqrt(x, guess):
    def closeEnough(x, guess):
        if math.fabs(square(guess) - x) > 0.01:
            return True
        else:
            return False

    def improveGuess(x, guess):
        return (((x / guess) + guess) / 2)

    while closeEnough(x, guess):
        guess = improveGuess(x, guess)
    return guess

def pythag(a, b):
    a2b2 = square(a) + square(b)
    return sqrt(a2b2, 1.0)

print pythag(2, 3)
~
```

› Our final code for finding the longest side of a triangle is longer than what we had to start, but it's more readable, more robust, and generally better.

```
def sqrt(x, guess):
    def closeEnough(x, guess):
        ...
    def improveGuess(x, guess):
        ...
...
```

These functions are now visible only to code within the **sqrt()** definition – we say they're in the scope of **sqrt()**. Anything outside of it has no idea that they even exist. This way, if we later need to define similar functions for improving a guess in a different context, we won't face the issue of colliding names or the headache of figuring out what **improveGuess1()** and **improveGuess2()** do.

Layers of abstraction

Hopefully, this example has demonstrated how powerful a technique abstraction is. Bear in mind that there are many layers of abstraction present in everything you do on a computer that you never think of.

For instance, when you're programming do you know how Python represents integers in the computer's memory? Or how the CPU performs arithmetic operations such as addition and subtraction?

The answer is probably no. You just accept the fact that typing **2 + 3** in to the Python interpreter returns the correct result, and you never have to worry about how it does this. You treat it as a black box.

Think how much longer it would take you to program if you had to manually take care of what data went in which memory location, to work with binary numbers, and translate alphabetic characters in to their numeric representations – thank goodness for abstraction! ■

Files and modules done quickly

It's time to expand your library of functions and grab external data with just two simple lines of Python.

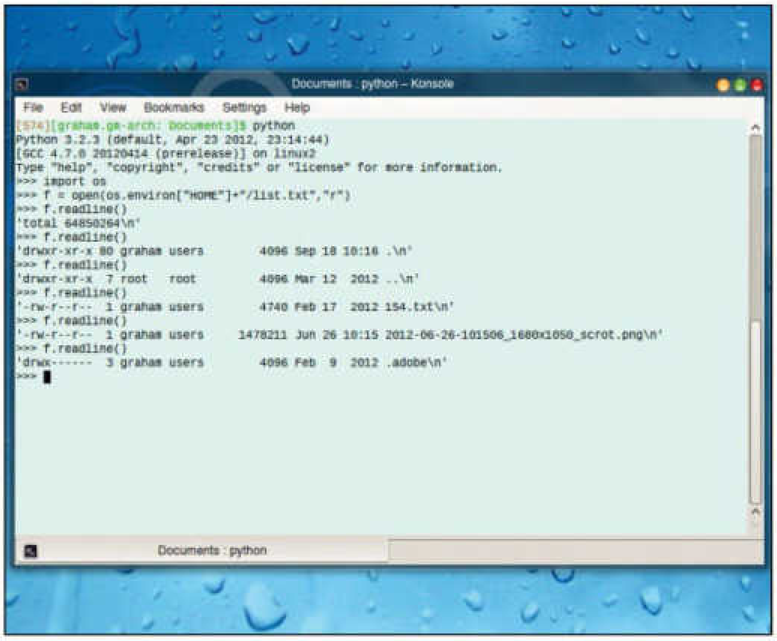
For the majority of programming projects, you don't get far before facing the age-old problem of how to get data into and out of your application. Whether it's using punched cards to get patterns into a 19th century Jacquard textile loom, or Google's bots skimming websites for data to feed its search engine, dealing with external input is as fundamental as programming itself.

And it's a problem and a concept that you may be more familiar with on the command line. When you type **ls** to list the contents of the current directory, for example, the command is reading in the contents of a file, the current directory, and then outputting the contents to another, the terminal.

Of course, the inputs and outputs aren't files in the sense most people would recognise, but that's the way the Linux filesystem has been designed – nearly everything is a file. This helps when you want to save the output of a command, or use that output as the input to another.

You may already know that typing **ls >list.txt** will redirect the output from the command to a file called **list.txt**, but you can take this much further because the output can be treated exactly like a file. **ls | sort -r** will pipe (that's the vertical bar character) the output of **ls** into the input of **sort** to create a reversed alphabetical list of a folder's contents. The complexity of how data input and output can be accomplished is entirely down to your programming environment. Every language will include functions to load and save data, for instance, but this can either be difficult or easy depending on how many assumptions the language is willing to make on your behalf. However, there's always a logical sequence of events that need to occur.

› **When you read a file, most languages will step through its data from the beginning to the end in chunks you specify. In this example, we're reading a line at a time.**



```
[574][graham.gm-arch: Documents]$ python
Python 3.2.3 (default, Apr 23 2012, 23:14:44)
[GCC 4.7.0 20120414 (prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import os
>>> f = open(os.environ["HOME"]+"/list.txt","r")
>>> f.readline()
'total 64850264\n'
>>> f.readline()
'drwxr-xr-x 80 graham users       4096 Sep 18 10:16 .\n'
>>> f.readline()
'drwxr-xr-x  7 root   root        4096 Mar 12  2012 ..\n'
>>> f.readline()
'-rw-r--r--  1 graham users       4740 Feb 17  2012 154.txt\n'
>>> f.readline()
'-rw-r--r--  1 graham users    1478211 Jun 26 10:15 2012-06-26-101506_1680x1050_scrot.png\n'
>>> f.readline()
'drwx------  3 graham users       4096 Feb  9  2012 .adobe\n'
>>> 
```

Documents : python – Konsole
File Edit View Bookmarks Settings Help
Documents : python

You will first need to open a file, creating one if it doesn't exist, and then either read data from this file, or write data to it, before explicitly closing the file again so that other processes can use it.

Most languages require you to specify a read mode when you open a file, because this tells the filesystem whether to expect file modifications or not. This is important because many different processes may also want to access the file, and if the filesystem knows the file is being changed, it won't usually allow access. However, many processes can access a read-only file without worrying about the integrity of the data it holds, because nothing is able to change it. If you know about databases, it's the same kind of problem you face with multiple users accessing the same table.

> **"If the filesystem knows the file is being changed, it won't allow access"**

In Python, as with most other languages, opening a file to write or as read-only can be done with a single line:

```
>>> f = open("list.txt", "r")
```

If the file doesn't exist, Python will generate a "No such file or directory" error. To avoid this, we've used the output from our command line example to create a text file called **list.txt**. This is within the folder from where we launched the Python interpreter.

Environment variables

Dealing with paths, folders and file locations can quickly become complicated, and it's one of the more tedious issues you'll face with your own projects. You'll find that different environments have different solutions for finding files, with some creating keywords for common locations and others leaving it to the programmer.

This isn't so bad when you only deal with files created by your projects, but it becomes difficult when you need to know where to store a configuration file or load a default icon. These locations may be different depending on your Linux distribution or desktop, but with a cross-platform language such as Python, they'll also be different for each operating system. For that reason, you might want to consider using environment variables. These are similar to variables with a global scope in many programming languages, but they apply to any one user's Linux session rather than within your own code. If you type **env** on the command line, for instance, you'll see a list of the environmental variables currently set for your terminal session. Look closely, and you'll see a few that apply to default locations and, most importantly, one called **HOME**. The value assigned to this environmental variable will be the location of your home folder on your Linux system, and if we want to use this within our Python script, we first need to add a line to import the operating system-specific module. The line to do this is:

```
import os
```

This command is also opening a file, but not in the same way we opened **list.txt**. This file is known as a module in Python terms, and modules like this 'import' functionality, including statements and definitions, so that a programmer doesn't have to keep re-inventing the wheel.

Modules extend the simple constructs of a language to add portable shortcuts and solutions, which is why other languages might call them libraries. Libraries and modules are a little like copying and pasting someone's own research and insight into your own project. Only it's better than that, because modules such as **os** are used by everyone, turning the way they do things into a standard.

Setting the standard

There are even libraries called **std**, and these embed standard ways of doing many things a language doesn't provide by default, such as common mathematical functions, data types and string services, as well as file input/output and support for specific file types. You will find the documentation for what a library does within an API. This will list each function, what it does, and what it requires as an input and an output. You should also be able to find the source files used by the import (and by **#include** in other languages). On most Linux systems, for example, **/lib/python2.x** will include all the modules. If you load **os.py** into a text editor, you'll see the code you've just added to your project, as well as which functions are now accessible to you.

There are many, many different modules for Python – it's one of the best reasons to choose it over any other language, and more can usually be installed with just a couple of clicks from your package manager.

But this is where the ugly spectre of dependencies can start to have an effect on your project, because if you want to give your code to someone else, you need to make sure that person has also got the same modules installed.

> **"If you load os.py into a text editor, you'll see the code you've just added"**

If you were programming in C or C++, for example, where your code is compiled and linked against binary libraries, those binary libraries will also need to be present on any other system that runs your code. They will become dependencies for your project, which is what package managers do when you install a complex package.

The os module

Getting back to our project, the **os** module is designed to provide a portable way of accessing operating system-dependent functionality so that you can write multi-platform applications without worrying about where files should be placed. This includes knowing where your home directory might be. To see what we mean, add the following piece of code to your project:

```
f = open(os.environ["HOME"]+"/list.txt","r")
```

This line will open the file **list.txt** in your home folder. Python knows which home folder is yours, because the **os.environ** function from the **os** module returns a string from an environmental variable, and the one we have asked it to return is **HOME**. But all we've done is open the file, we've not yet read any of its contents. This might seem counter-intuitive, but it's an historical throwback to the way that files used to be stored, which is why this is also the way nearly all languages work. It's only after a file has been opened that you can start to read its contents:

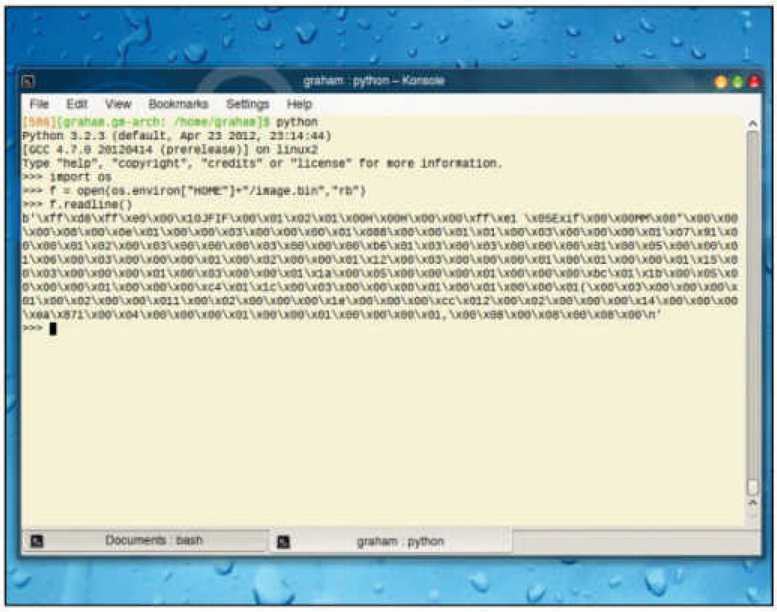
```
f.readline()
```

The above instruction will read a single line of the text file and output this to the interpreter as a string. Repeating the command will read the next line, because Python is remembering how far through the file it has read. Internally, this is being done using something called a pointer and this, too, is common to the vast majority of languages. Alternatively, if you wanted to read the entire file, you could use **f.read()**. Because our file contains only text, copying the contents to a Python string is an easy conversion. The same isn't true of a binary file. Rather than being treated as text, the organisation of the bits and bytes that make up a binary file are organised according to the file type used by the file – or no file type at all if it's raw data. As a result, Python (or any other programming language) would be unable to extract any context from a binary file, causing an error if you try to read it into a string. The solution, at least for the initial input, is to add a **b** flag when you first open the file, because this warns Python to expect raw binary. When you then try to read the input, you'll see the hexadecimal values of the file output to the display.

To make this data useful, you'll need to do some extra work, which we'll look at next; but first, make sure you close the open file, as this should ensure the integrity of your filesystem. As you might guess, the command looks like this:

```
f.close
```

And it's as easy as that! ■


graham : python – Konsole
File Edit View Bookmarks Settings Help
Documents : bash
graham : python

```
[586][graham.gs-arch: /home/graham]$ python
Python 3.2.3 (default, Apr 23 2012, 23:14:44)
[GCC 4.7.0 20120414 (prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import os
>>> f = open(os.environ["HOME"]+"/image.bin","rb")
>>> f.readline()
b'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x02\x01\x00H\x00H\x00\x00\xff\xe1 \x05Exif\x00\x00MM\x00*\x00\x00
\x00\x08\x00\x0e\x01\x00\x00\x03\x00\x00\x00\x01\x088\x00\x00\x01\x01\x00\x03\x00\x00\x00\x01\x07\x81\x0
0\x00\x01\x02\x00\x03\x00\x00\x00\x03\x00\x00\x00\xb6\x01\x03\x00\x03\x00\x00\x00\x01\x00\x05\x00\x00\x0
1\x06\x00\x03\x00\x00\x00\x01\x00\x02\x00\x00\x01\x12\x00\x03\x00\x00\x00\x01\x00\x01\x00\x00\x01\x15\x0
0\x03\x00\x00\x00\x01\x00\x03\x00\x00\x01\x1a\x00\x05\x00\x00\x00\x01\x00\x00\x00\xbc\x01\x1b\x00\x05\x0
0\x00\x00\x01\x00\x00\x00\xc4\x01\x1c\x00\x03\x00\x00\x00\x01\x00\x01\x00\x00\x01(\x00\x03\x00\x00\x00\x
01\x00\x02\x00\x00\x011\x00\x02\x00\x00\x00\x1e\x00\x00\x00\xcc\x012\x00\x02\x00\x00\x00\x14\x00\x00\x00
\xea\x871\x00\x04\x00\x00\x00\x01\x00\x00\x01\x00\x00\x00\x01,\x00\x08\x00\x08\x00\x08\x00\n'
>>> ▮
```

› **Binary files have no context without an associated file type and a way of handling them. Which is why you get the raw data output when you read one.**

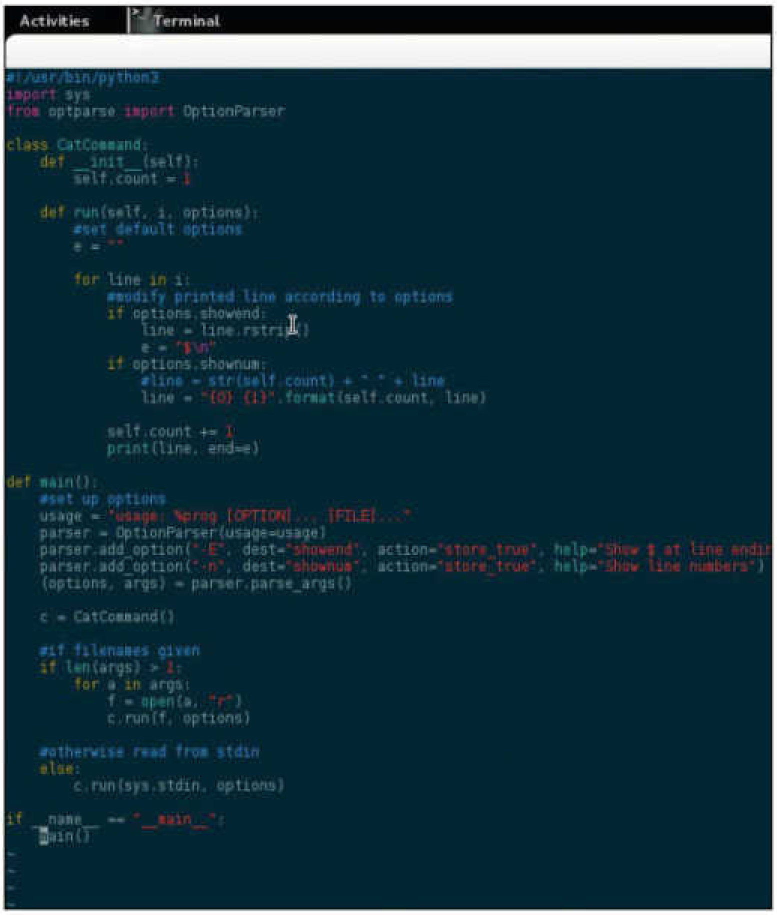
Write your own UNIX program

Try re-implementing classic Unix tools to bolster your Python knowledge and learn how to build real programs.

In the next few pages, that's what we're aiming to do: get you writing real programs. Over the next few tutorials, we're going to create a Python implementation of the popular Unix tool **cat**. Like all Unix tools, **cat** is a great target because it's small and focused on a single task, while using different operating system features, including accessing files, pipes and so on.

This means it won't take too long to complete, but will also expose you to a selection of Python's core features in the Standard Library, and once you've mastered the basics, it's learning the ins-and-outs of your chosen language's libraries that will let you get on with real work.

> "You now know more than enough to start writing real programs"



```
#!/usr/bin/python3
import sys
from optparse import OptionParser

class CatCommand:
    def __init__(self):
        self.count = 1

    def run(self, i, options):
        #set default options
        e = ""

        for line in i:
            #modify printed line according to options
            if options.showend:
                line = line.rstrip()
                e = "$\n"
            if options.shownum:
                #line = str(self.count) + " " + line
                line = "{0} {1}".format(self.count, line)

            self.count += 1
            print(line, end=e)

def main():
    #set up options
    usage = "usage: %prog [OPTION]... [FILE]..."
    parser = OptionParser(usage=usage)
    parser.add_option("-E", dest="showend", action="store_true", help="Show $ at line endi
    parser.add_option("-n", dest="shownum", action="store_true", help="Show line numbers")
    (options, args) = parser.parse_args()

    c = CatCommand()

    #if filenames given
    if len(args) > 1:
        for a in args:
            f = open(a, "r")
            c.run(f, options)

    #otherwise read from stdin
    else:
        c.run(sys.stdin, options)

if __name__ == "__main__":
    main()
~
~
~
~
```

The final program we'll be implementing. It's not long, but it makes use of a lot of core language features you'll be able to re-use time and again.

Our goal for the project overall is to:

- Create a Python program, **cat.py**, that when called with no arguments accepts user input on the standard input pipe until an end of line character is reached, at which point it sends the output to standard out.
- When called with file names as arguments, **cat.py** should send each line of the files to standard output, displaying the whole of the first file, then the whole of the second file.
- It should accept two arguments: **-E**, which will make it put **$** signs at the end of each line; and **-n**, which will make it put the current line number at the beginning of each line.

This time, we're going to create a **cat** clone that can work with any number of files passed to it as arguments on the command line. We're going to be using Python 3, so if you want to follow along, make sure you're using the same version, because some features are not backwards-compatible with Python 2.x.

Python files

Let's start with the easiest part of the problem: displaying the contents of a file, line by line, to standard out. In Python, you access a file with the **open** function, which returns a file-object that you can later read from, or otherwise manipulate. To capture this file-object for use later in your program, you need to assign the result of running the **open** function to a variable, like so:

```
file = open("hello.txt", "r")
```

This creates a variable, **file**, that will later allow us to read the contents of the file **hello.txt**. It will only allow us to read from this file, not write to it, because we passed a second argument to the open function, **r**, which specified that the file should be opened in read-only mode.

With access to the file now provided through the newly-created **file** object, the next task is to display its contents, line by line, on standard output. This is very easy to achieve, because in Python files are iterable objects.

Iterable objects, such as lists, strings, tuples and dictionaries, allow you to access their individual member elements one at a time through a **for** loop. With a file, this means you can access each line contained within simply by putting it in a **for** loop, as follows:

```
for line in file:
        print(line)
```

The **print** function then causes whatever argument you pass to it to be displayed on standard output.

If you put all this in a file, make it executable and create a **hello.txt** file in the same directory, you'll see that it works rather well. There is one oddity, however – there's an empty line between each line of output.

The reason this happens is that **print** automatically adds a newline character to the end of each line. Because there's already a newline character at the end of each line in **hello.txt** (there is, even if you can't see it, otherwise everything would be on one line!), the second newline character leads to an empty line.

You can fix this by calling **print** with a second, named argument such as: **print(line, end="")**. This tells **print** to put an empty string, or no character, at the end of each line instead of a newline character.

Passing arguments

This is all right, but compared to the real **cat** command, there's a glaring omission here: we would have to edit the program code itself to change which file is being displayed to standard out. What we need is some way to pass arguments on the command line, so that we could call our new program by typing **cat.py hello.txt** on the command line. Since Python has 'all batteries included', this is a fairly straightforward task, as well.

The Python interpreter automatically captures all arguments passed on the command line, and a module called **sys**, which is part of the Standard Library, makes this available to your code.

Even though **sys** is part of the Standard Library, it's not available to your code by default. Instead, you first have to import it to your program and then access its contents with dot notation – don't worry, we'll explain this in a moment. First, to import it to your program, add:

```
import sys
```

to the top of your **cat.py** file.

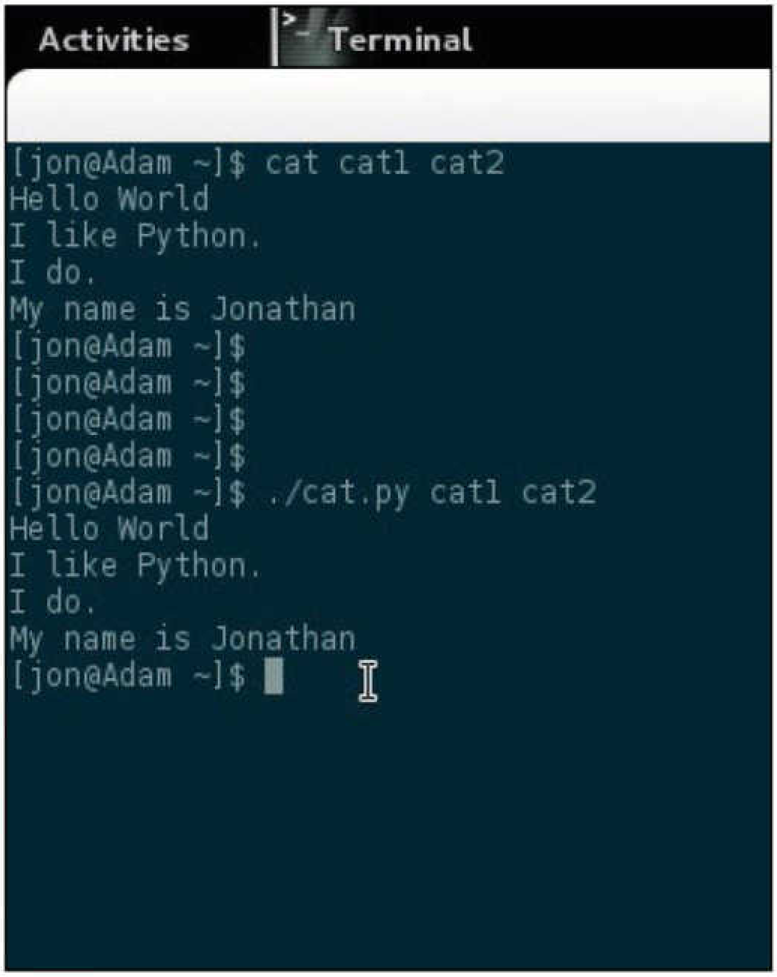
The part of the **sys** module that we're interested in is the **argv** object. This object stores all of the arguments passed on the command line in a Python list, which means you can access and manipulate it using various techniques we've seen in previous tutorials and will show in future ones.

There are only two things you really need to know about this. They are:

- The first element of the list is the name of the program itself – all arguments follow this.
- To access the list, you need to use dot notation – that is to say, **argv** is stored within **sys**, so to access it, you need to type **sys.argv**, or **sys.argv[1]** to get the first argument to your program.

Knowing this, you should now be able to adjust the code we created previously by replacing **hello.txt** with **sys.argv[1]**. When you call **cat.py** from the command line, you can then pass the name of any text file, and it will work just the same.

> "The part of the sys module we're interested in is the argv object"



```
[jon@Adam ~]$ cat cat1 cat2
Hello World
I like Python.
I do.
My name is Jonathan
[jon@Adam ~]$
[jon@Adam ~]$
[jon@Adam ~]$
[jon@Adam ~]$
[jon@Adam ~]$ ./cat.py cat1 cat2
Hello World
I like Python.
I do.
My name is Jonathan
[jon@Adam ~]$
```

The output of the real Unix command, cat, and our Python re-implementation, are exactly the same in this simple example.

Many files

Of course, our program is meant to accept more than one file and output all their contents to standard output, one after another, but as things stand, our program can only accept one file as an argument.

To fix this particular problem, you need to loop over all the files in the **argv** list. The only thing that you need to be careful of when you do this is that you exclude the very first element, because this is the name of the program itself. If you think back to our previous article on data types and common list operations, you'll realise this is easily done with a slice. This is just one line:

```
for file in sys.argv[1:]:
```

Because operating on all the files passed as arguments to a program is such a common operation, Python provides a shortcut for doing this in the Standard Library, called **fileinput**.

In order to use this shortcut, you must first import it by putting **import fileinput** at the top of your code. You will then be able to use it to recreate the rest of our **cat** program so far, as follows:

```
for line in fileinput.input():
            print(line, end="")
```

This simple shortcut function takes care of opening each file in turn and making all their lines accessible through a single iterator.

That's about all that we have space for in this tutorial. Although there has not been much code in this particular example, we hope you have started to get a sense for how much is available in Python's Standard Library (and therefore how much work is available for you to recycle), and how a good knowledge of its contents can save you a lot of work when implementing new programs. ■

Enhance your UNIX program

Our tour of the Python programming language continues, as we continue our clone of the Unix **cat** command.

The previous tutorial showed you how to build a simple **cat** clone in Python. In this guide, we're going to add some more features to our program, including the ability to read from the standard input pipe, just like the real **cat**, and the ability to pass options to your **cat** clone. So, without further delay, let's dive in.

Fortunately, you already know everything you need to interact with the standard input pipe. In Linux, all pipes are treated like files – you can pass a file as an argument to a command, or you can pass a pipe as an argument. It doesn't matter which you do, because they're basically the same thing.

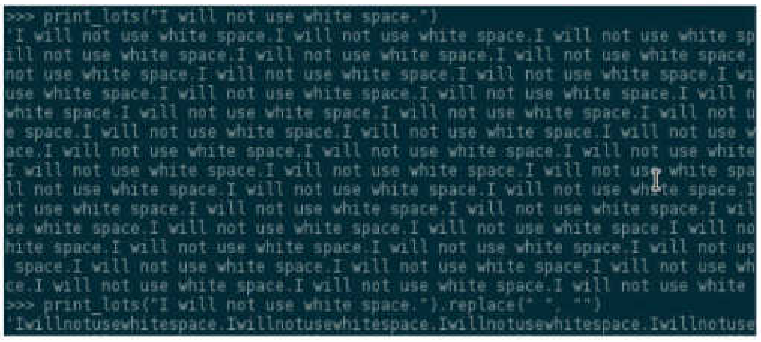
> **“Python provides us with a much more powerful alternative to sys.argv”**

In Python, the same is true. All you need to get to work with the standard input pipe is access to the sys library, which if you followed along last time, you already have. Let's write a little sample program first to demonstrate:

```
import sys
for line in sys.stdin:
        print(line, end=””)
```

The first line imports the **sys** module. The lines that follow are almost identical to those that we had last time. Rather than specifying the name of a file, however, we specified the name of the file-object, **stdin**, which is found inside the **sys** module. Just like a real file, in Python the standard input pipe is an iterable object, so we use a **for** loop to walk through each line.

You might be wondering how this works, though, because standard input starts off empty. If you run the program, you will see what happens. Rather than printing out everything that's present straightaway, it will simply wait. Every time a new line character is passed to standard input (by pressing [Return]), it will then print everything that came before it to standard output.



```
>>> print_lots("I will not use white space.")
'I will not use white space.I will not use white space.I will not use white sp
ill not use white space.I will not use white space.I will not use white space.
not use white space.I will not use white space.I will not use white space.I wi
use white space.I will not use white space.I will not use white space.I will n
white space.I will not use white space.I will not use white space.I will not u
e space.I will not use white space.I will not use white space.I will not use w
ace.I will not use white space.I will not use white space.I will not use white
I will not use white space.I will not use white space.I will not use white spa
ll not use white space.I will not use white space.I will not use white space.I
ot use white space.I will not use white space.I will not use white space.I wil
se white space.I will not use white space.I will not use white space.I will no
hite space.I will not use white space.I will not use white space.I will not us
 space.I will not use white space.I will not use white space.I will not use wh
ce.I will not use white space.I will not use white space.I will not use white
>>> print_lots("I will not use white space.").replace(" ", "")
'Iwillnotusewhitespace.Iwillnotusewhitespace.Iwillnotusewhitespace.Iwillnotuse
```

The Python language comes with all the bells and whistles you need to write useful programs. In this example, you can see the replace method applied to a string in order to remove all white space – in the tutorial, we used the rstrip method for a similar purpose.

Right, now we have two modes that our program can operate in, but we need to put them together into a single program. If we call our program with arguments, we want it to work like last time – that is, by concatenating the files' contents together; if it's called without any arguments, we want our program to work by repeating each line entered into standard input. We could easily do this with what we have learned so far – simply check to see what the length of the **sys.argv** array is. If it's greater than 1, then do last lesson's version, otherwise do this version:

```
if len(sys.argv) > 1:
        [last month...]
else:
        [this month...]
```

Pretty straightforward. The only point of interest here is the use of the **len()** function, seeing as we're on a journey to discover different Python functions. This function is built in to Python, and can be applied to any type of sequence object (a string, tuple or list) or a map (like a dictionary), and it always tells you how many elements are in that object.

There are more useful functions like this, which you can find at **http://docs.python.org/3/library/functions.html**.

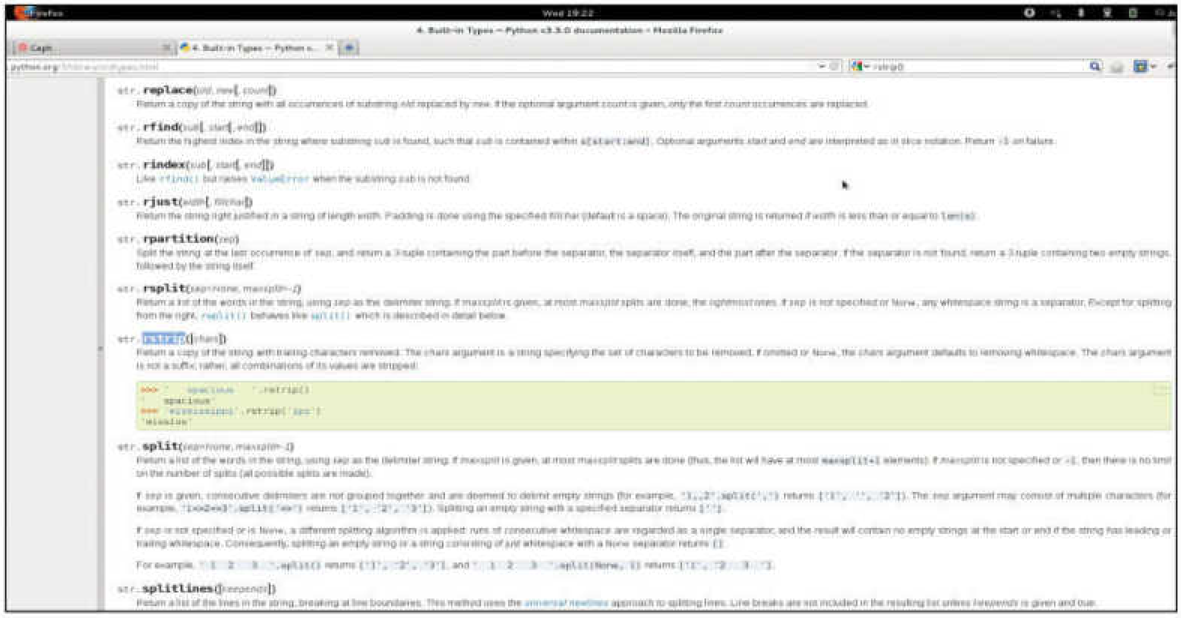
Parsing arguments and options

This is quite a simplistic approach, however, and Python actually provides us with a much more powerful alternative to **sys.argv**. To demonstrate this, we are going to add two options to our program that will modify the output generated by our program.

You may not have realised it, but **cat** does in fact have a range of options. We're going to implement the **-E**, which shows dollar symbols at the end of lines, and **-n**, which displays line numbers at the beginning of lines.

To do this, we'll start by setting up an **OptionParser**. This is a special object, provided as part of the **optparse** module, which will do most of the hard work for you. As well as automatically detecting options and arguments, saving you a lot of hard work, **OptionParser** will automatically generate help text for your users in the event that they use your program incorrectly or pass **--help** to it, like this:

```
[jon@LT04394 ~]$ ./cat.py --help
Usage: cat.py [OPTION]... [FILE]...
```


Firefox
Wed 19:22
4. Built-in Types — Python v3.3.0 documentation - Mozilla Firefox
Capt...
4. Built-in Types — Python v...
python.org/3/library/stdtypes.html
rstrip()

str.**replace**(old, new[, count])
Return a copy of the string with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

str.**rfind**(sub[, start[, end]])
Return the highest index in the string where substring sub is found, such that sub is contained within s[start:end]. Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

str.**rindex**(sub[, start[, end]])
Like rfind() but raises ValueError when the substring sub is not found.

str.**rjust**(width[, fillchar])
Return the string right justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than or equal to len(s).

str.**rpartition**(sep)
Split the string at the last occurrence of sep, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

str.**rsplit**(sep=None, maxsplit=-1)
Return a list of the words in the string, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done, the rightmost ones. If sep is not specified or None, any whitespace string is a separator. Except for splitting from the right, rsplit() behaves like split() which is described in detail below.

str.**rstrip**([chars])
Return a copy of the string with trailing characters removed. The chars argument is a string specifying the set of characters to be removed. If omitted or None, the chars argument defaults to removing whitespace. The chars argument is not a suffix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

str.**split**(sep=None, maxsplit=-1)
Return a list of the words in the string, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done (thus, the list will have at most maxsplit+1 elements). If maxsplit is not specified or -1, then there is no limit on the number of splits (all possible splits are made).

If sep is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, '1,,2'.split(',') returns ['1', '', '2']). The sep argument may consist of multiple characters (for example, '1<>2<>3'.split('<>') returns ['1', '2', '3']). Splitting an empty string with a specified separator returns [''].

If sep is not specified or is None, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a None separator returns [].

For example, ' 1 2   3 '.split() returns ['1', '2', '3'], and ' 1 2   3 '.split(None, 1) returns ['1', '2   3 '].

str.**splitlines**([keepends])
Return a list of the lines in the string, breaking at line boundaries. This method uses the universal newlines approach to splitting lines. Line breaks are not included in the resulting list unless keepends is given and true.

The Python 3 website provides excellent documentation for a wealth of built-in functions and methods. If you ever wonder how to do something in Python, http://docs.python.org/3/ should be your first port of call.

```
Options:
  -h, --help  show this help message and exit
  -E          Show $ at line endings
  -n          Show line numbers
```

Just like a real program!

To get started with **OptionParser**, you first need to import the necessary components:

```
from optparse import OptionParser
```

You may notice that this looks a bit different from what we saw before. Instead of importing the entire module, we're only importing the **OptionParser** object. Next, you need to create a new instance of the object, add some new options for it to detect with the **add_option** method, and pass it a **usage** string to display:

```
usage = "usage: %prog [option]... [file]..."
parser = OptionParser(usage=usage)
parser.add_option("-E", dest="showend", action="store_
true", help="Show $ at line
endings")
parser.add_option("-n", dest="shownum", action="store_
true", help="Show line numbers")
```

The **%prog** part of the **usage** string will be replaced with the name of your program. The **dest** argument specifies what **name** you'll be able to use to access the value of an argument once the parsing has been done, while the **action** specifies what that value should be. In this case, the action **store_true** says to set the **dest** variable to **True** if the argument is present, and **False** if it is not. You can read about other actions at **http://docs.python.org/3/library/optparse.html**.

> "Don't confuse yourself by putting the variables the other way around!"

Finally, with everything set, you just need to parse the arguments that were passed to your program and assign the results to two array variables:

```
(options, args) = parser.parser_args()
```

The **options** variable will contain all user-defined options, such as **-E** or **-n**, while **args** will contain all positional arguments left over after parsing out the options. You can call these variables whatever you like, but the two will always be set in the same order, so don't confuse yourself by putting the variables the other way around! With the argument-parsing code written, you'll next want to start implementing the code that will run when a particular option is set. In both cases, we'll be modifying the string of text that's output by the program, which means you'll need to know a little bit about Python's built-in string editing functions. Let's think about the **-E**, or **showend**, option first. All we want this to do is remove the invisible line break that's at the end of every file (or every line of the standard input pipe, as implied by pressing [Return]), and replace it with a dollar symbol followed by a line break.

The first part, removing the existing new line, can be achieved by the **string.rstrip()** method. This removes all white space characters by default, at the right-hand edge of a string. If you pass a string to it as an argument, it will strip those characters from the right-hand edge instead of white space. In our case, just white space will do.

Completing the job

The second part of the job is as simple as setting the end variable in the **print** statement to the string **$\n** and the job is almost complete. We say "almost complete" because we still need to write some more logic to further control the flow of the program based on what options were set, as well as whether or not any arguments are passed. The thing is, this logic needs to be a little more complicated than it ordinarily would be because we need to maintain a cumulative count of lines that have been printed as the program runs to implement the second **-n**, or **shownum**, option.

While there are several ways you could achieve this, in the next tutorial we're going to introduce you to a bit of object-oriented programming in Python and implement this functionality in a class. We'll also introduce you to a very important Python convention – the **main()** function and the **name** variable. In the meantime, you can keep yourself busy by investigating the **string.format()** method and see whether you can figure out how you can append a number to the beginning of each line. ■

Finish up your UNIX program

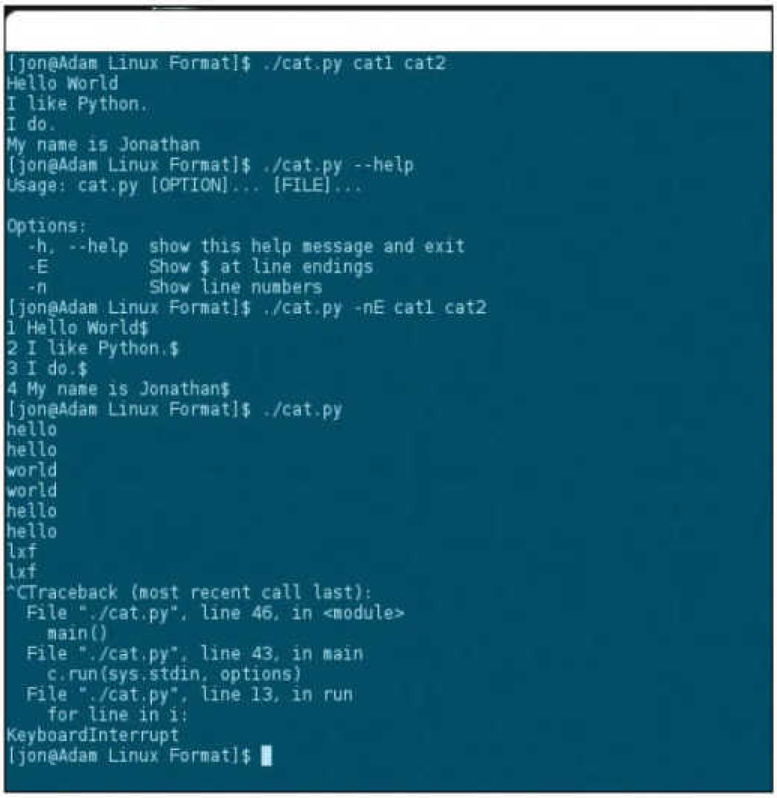
Our guide to the Python programming language continues. This tutorial, we're going to finish our clone of cat.

We've come quite a long way over the last two tutorials, having implemented the ability to echo the contents of multiple files to the screen, the ability to echo standard input to the screen and the ability to detect and act upon options passed by the user of our program. All that remains is for us to implement the line number option and to gather together everything else we've written into a single, working program.

> "It's a very natural way of thinking because it mirrors the real world"

Objects

Last time, we ended by saying that there are many ways we could implement the line counting option in our program. We're going to show you how to do it in an object-oriented style, because it gives us an excuse to introduce you to this aspect of Python programming. You could, however, with a little careful thought, implement the same function with some nested **for** loops, although they're not nearly as readable as object-oriented code.



```
[jon@Adam Linux Format]$ ./cat.py cat1 cat2
Hello World
I like Python.
I do.
My name is Jonathan
[jon@Adam Linux Format]$ ./cat.py --help
Usage: cat.py [OPTION]... [FILE]...

Options:
  -h, --help  show this help message and exit
  -E          Show $ at line endings
  -n          Show line numbers
[jon@Adam Linux Format]$ ./cat.py -nE cat1 cat2
1 Hello World$
2 I like Python.$
3 I do.$
4 My name is Jonathan$
[jon@Adam Linux Format]$ ./cat.py
hello
hello
world
world
hello
hello
lxf
lxf
^CTraceback (most recent call last):
  File "./cat.py", line 46, in <module>
    main()
  File "./cat.py", line 43, in main
    c.run(sys.stdin, options)
  File "./cat.py", line 13, in run
    for line in i:
KeyboardInterrupt
[jon@Adam Linux Format]$ █
```

Just to prove that it works, here's our cat implementation, with all of the options being put to use.

When building complicated programs, figuring out how to organise them so they remain easy to read, easy to track which variables are being used by which functions, and easy to update, extend, or add new features, can be challenging. To make this easier, there are various paradigms that provide techniques for managing complexity.

One of these paradigms is the concept of object-oriented programming. In object-oriented programming, the elements of the program are broken down into objects which contain state – variables, in other words – that describe the current condition of the object, and methods, that allow us to perform actions on those variables or with that object.

It's a very natural way of thinking, because it mirrors the real world so closely. We can describe a set of properties about your hand, such as having five fingers that are in certain locations, and we can describe certain methods or things you can do with your hand, such as moving one finger to press a key, or holding a cup. Your hand is an object, complete with state and methods that let you work with it.

We're going to turn our **cat** program into an object, where its state records how many lines have been displayed, and its methods perform the action of the **cat** program – redisplaying file contents to the screen.

Python objects

Python implements objects through a class system. A class is a template, and an object is a particular instance of that class, modelled on the template. We define a new class with a keyword, much like we define a new function:

```
class catCommand:
```

Inside the class, we specify the methods (functions) and state that we want to associate with every instance of the object. There are some special methods, however, that are frequently used. One of these is the **init** method. This is run when the class is first instantiated into a particular object, and allows you to set specific variables that you want to belong to that object.

```
def __init__(self):
    self.count = 1
```

In this case, we've assigned **1** to the **count** variable, and we'll be using this to record how many lines have been displayed. You probably noticed the **self** variable, passed as the first argument to the method, and wondered what on earth that was about. Well, it is the main distinguishing feature between methods and ordinary functions.

Methods, even those which have no other arguments, must have the **self** variable. It is an automatically populated variable, which will always point to the particular instance of the object that you're working with. So **self.count** is a count variable that is exclusive to individual instances of the **catCommand** object.

The run method

We next need to write a method that will execute the appropriate logic depending on whether certain options are set. We've called this the **run** method:

```
def run(self, i, options):
    #set default options
    e = “”
    for line in i:
        #modify printed line according to options
        if options.showend:
            [...last time]
        if options.shownum:
            line = “{0} {1}”.format(self.count, line)
        self.count += 1
        print(line, end=e)
```

Notice that we've passed the **self** variable to this method, too. The two other arguments passed to this function are arguments that we'll pass when we call the method later on, just like with a normal function. The first, **i**, is going to be a reference to whichever file is being displayed at this moment, while the **options** variable is a reference to the options decoded by the **OptParse** module.

The logic after that is clear – for each line in the current file, modify the line depending on what options have been set. Either we do as last time, and modify the end character to be **"$\n"** or we modify the line, using the **.format** method that we suggested you research last time, to append the **count** variable, defined in the **init** method, to the rest of the line. We then increment the count and print the line.

The most important part is the use of **self**. It lets us refer to variables stored within the current instance of the object. Because it's stored as part of the object, it will persist after the current execution of the **run** method ends. As long as we use the **run** method attached to the same object each time we **cat** a new file in the argument list, the count will remember how many lines were displayed in the last file, and continue to count correctly.

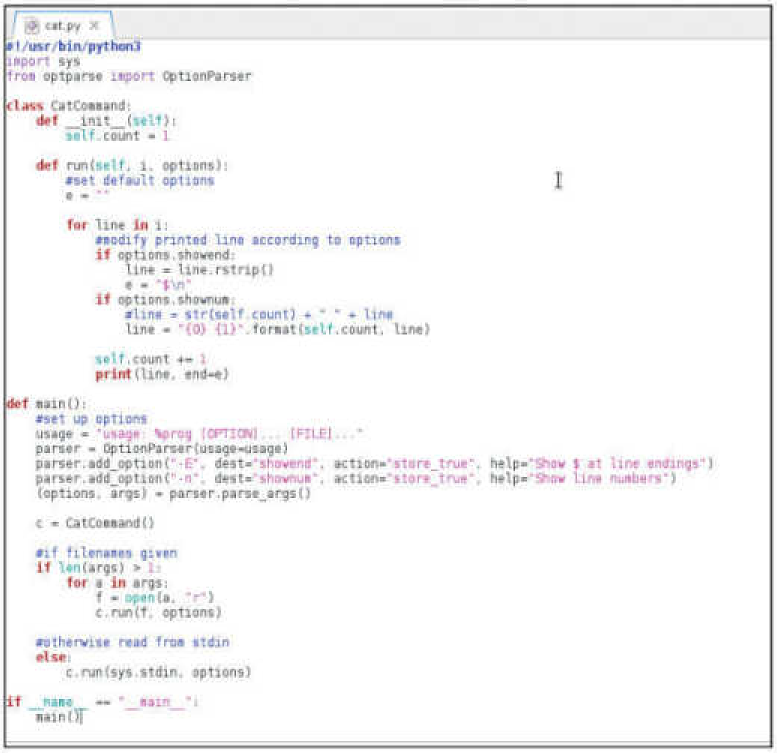
It might seem more natural – given the description of methods as individual actions that can be taken by our objects – to split each argument into a different method, and this is a fine way to approach the problem.

> **“The last thing to do is call the main function when the program runs”**

The reason we've done it this way, though, is that we found it meant we could re-use more code, making it more readable and less error-prone.

Now all that's left to do is to tie everything together. We are going to do this by writing a main function. This isn't actually required in Python, but many programs follow this idiom, so we will too:

```
def main():
    [option parsing code ...]
    c = catCommand()
    if len(args) > 1:
        for a in args:
            f = open(a, “r”)
            c.run(f, options)
    else:
        c.run(sys.stdin, options)
```



```
#!/usr/bin/python3
import sys
from optparse import OptionParser

class CatCommand:
    def __init__(self):
        self.count = 1

    def run(self, i, options):
        #set default options
        e = ""

        for line in i:
            #modify printed line according to options
            if options.showend:
                line = line.rstrip()
                e = "$\n"
            if options.shownum:
                #line = str(self.count) + " " + line
                line = "{0} {1}".format(self.count, line)

            self.count += 1
            print(line, end=e)

def main():
    #set up options
    usage = "usage: %prog [OPTION]... [FILE]..."
    parser = OptionParser(usage=usage)
    parser.add_option("-E", dest="showend", action="store_true", help="Show $ at line endings")
    parser.add_option("-n", dest="shownum", action="store_true", help="Show line numbers")
    (options, args) = parser.parse_args()

    c = CatCommand()

    #if filenames given
    if len(args) > 1:
        for a in args:
            f = open(a, "r")
            c.run(f, options)

    #otherwise read from stdin
    else:
        c.run(sys.stdin, options)

if __name__ == "__main__":
    main()
```

The completed program isn't very long, but it has given us a chance to introduce you to many different aspects of the Python language.

We haven't filled in the object parsing code from the previous tutorial, because that hasn't changed. What is new is the **c = catCommand()** line. This is how we create an instance of a class – how we create a new object. The **c** object now has a variable, **count**, which is accessible by all its methods as the **self.count** variable. This is what will enable us to track the line numbers.

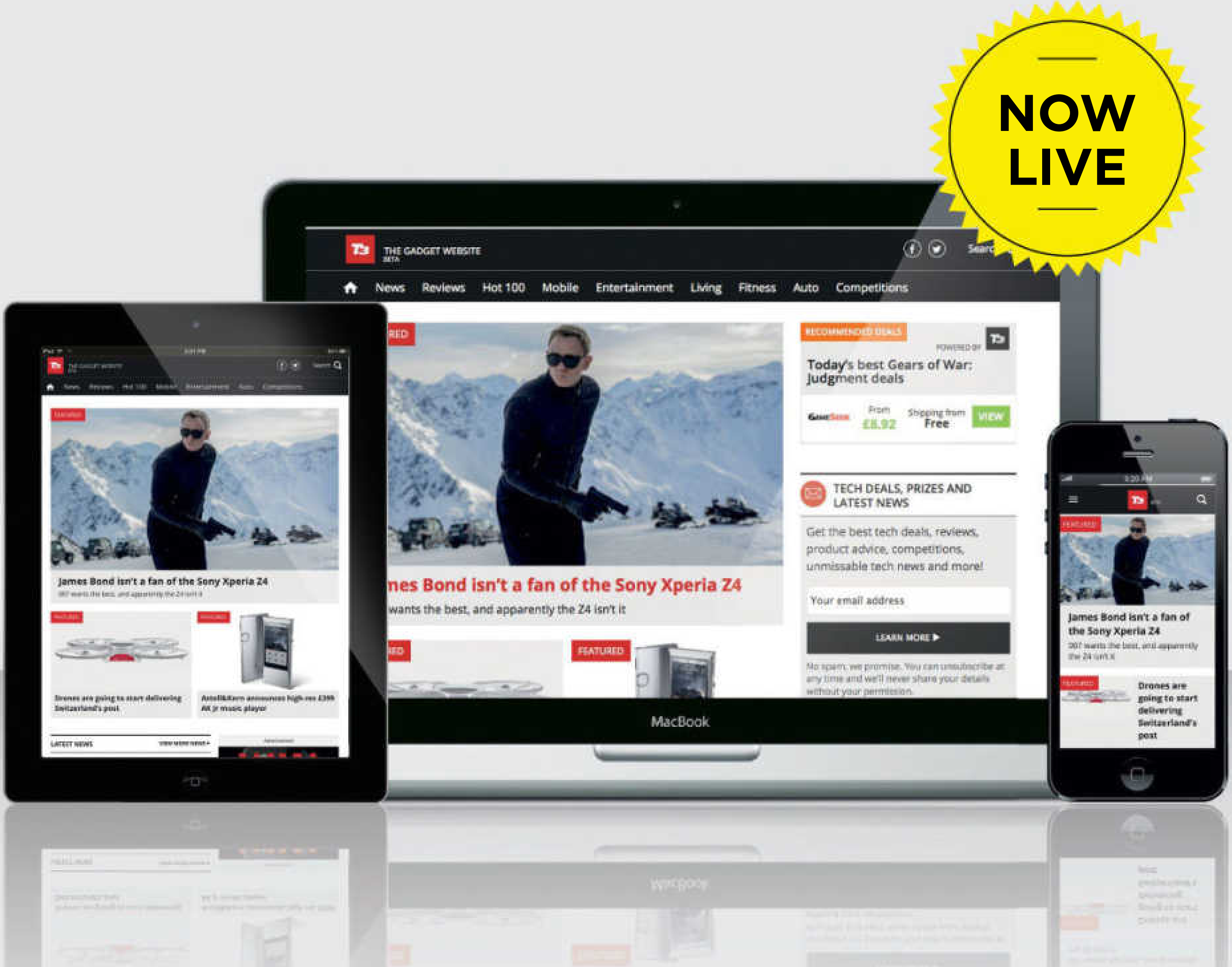
We then check to see whether any arguments have been passed. If they have, then we call the **run** method of the object **c** for each file that was passed as an argument, passing in any options extracted by **OptParse** along the way. If there weren't any arguments, though, we simply call the **run** method with **sys.stdin** instead of a file object.

The final thing we need to do here is actually call the main function when the program is run:

```
if __name__ == “__                main__”:
        main()
```

These last two lines are the strangest of all, but quite useful in a lot of circumstances. The **name** variable is special – when the program is run on the command line, or otherwise as a standalone application, it is set to **main**; however, when it is imported as an external module to other Python programs, it's not.

In this way, we can automatically execute **main** when run as a standalone program, but not when we're importing it as a module. ■


NOW LIVE
THE GADGET WEBSITE
News
Reviews
Hot 100
Mobile
Entertainment
Living
Fitness
Auto
Competitions
FEATURED
James Bond isn't a fan of the Sony Xperia Z4
007 wants the best, and apparently the Z4 isn't it
RECOMMENDED DEALS
Today's best Gears of War: Judgment deals
From £8.92
Shipping from Free
VIEW
TECH DEALS, PRIZES AND LATEST NEWS
Get the best tech deals, reviews, product advice, competitions, unmissable tech news and more!
Your email address
LEARN MORE ▶
No spam, we promise. You can unsubscribe at any time and we'll never share your details without your permission.
Drones are going to start delivering Switzerland's post
Astell&Kern announces high-res £399 AK Jr music player
LATEST NEWS
MacBook

Neater code with modules

Untangle the horrible mess of code that you've made and add coherent structure to your programs.

In previous tutorials, we have mentioned how programming is all about managing complexity, and we've introduced you to quite a few of the tools and techniques that help programmers do this. From variables to function definitions or object-orientation, they all help. One tool we've yet to cover, in part because you don't start to come across it until you're writing larger programs, is the idea of modules and name spaces.

Yet, if you've written a program of any length, even just a few hundred lines, this is a tool you're no doubt desperate for. In a long file of code, you'll have noticed how quickly it becomes more difficult to read. Functions seem to blur into one another, and when you're trying to hunt down the cause of the latest error, you find it difficult to remember exactly where you defined that all-important variable.

These problems are caused by a lack of structure. With all your code in a single file, it's more difficult to determine the dependencies between elements of your program – that is, which parts rely on each other to get work done – and it's more difficult to visualise the flow of data through your program.

> **"As your programs grow in length, there are other problems that occur"**

As your programs grow in length, too, there are other problems that begin to occur: for instance, you may find yourself with naming conflicts, as two different parts of your program require functions called **add** (for example, adding integers or adding fractions in a mathematics program), or you may have written a useful function that you want to share with other programs that you're writing, and the only tool you have to do that is boring and error-prone copy and pasting.

Untangling the mess

Modules are a great way to solve all of these problems, enabling you to put structure back into your code, letting you avoid naming conflicts, and making it easier for you to share useful chunks of code between programs.

You've no doubt been using them all the time in your code as you've relied on Python built-in or third-party modules to provide lots of extra functionality. As an example, remember the **optparse** module we used before.

We included it in our program along with the **import** statement, like so:

```
import optparse
```

After putting this line at the top of our Python program, we magically got access to a whole load of other functions that automatically parsed command line options. We could access them by typing the module's name, followed by the name of the function we wanted to execute:

```
optparse.OptionParser()
```

This was great from a readability perspective. In our **cat** clone, we didn't have to wade through lots of code about parsing command line arguments; instead we could focus on the code that dealt with the logic of echoing file contents to the screen and to the standard output pipe. What's more, we didn't have to worry about using names in our own program that might collide with those in the **optparse** module, because they were all hidden inside the **optparse** namespace, and reusing this code was as easy as typing **import optparse** – no messy copy and pasting here.

How modules work

Modules sound fancy, and you might think they're complicated, but – in Python at least – they're really just plain old files. You can try it out for yourself. Create a new directory and inside it create a **fact.py** file. Inside it, define a function to return the factorial of a given number:

```
def factorial(n):
    result = 1
    while n > 0:
        if n == 1:
            result *= 1
        else:
            result *= n
        n -= 1
    return n
```

Create a second Python file called **doMath.py**. Inside this, first import the module you just created and then execute the **factorial** function, printing the result to the screen:

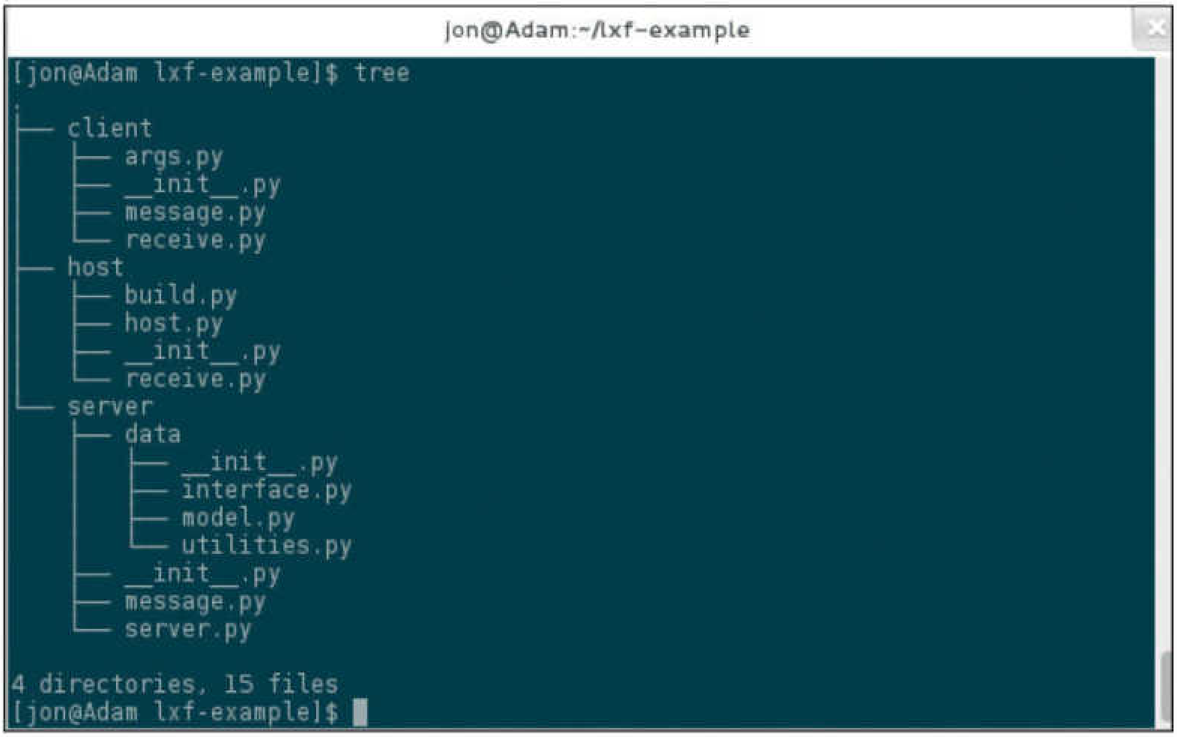
```
import fact
print fact.factorial(5)
```

Now, when you run the **doMath.py** file, you should see 120 printed on the screen. You should notice that the name of the module is just the name of the file, in the same directory, with the extension removed. We can then call any function defined in that module by typing the module's name, followed by a dot, followed by the function name.

The Python path

The big question that's left is, how does Python know where to look to find your modules?

The answer is that Python has a pre-defined set of locations that it looks in to find files that match the name specified in your import statements. It first looks inside all of the built-in modules, the location of which are defined when



```
jon@Adam:~/lxf-example
[jon@Adam lxf-example]$ tree
.
├── client
│   ├── args.py
│   ├── __init__.py
│   ├── message.py
│   └── receive.py
├── host
│   ├── build.py
│   ├── host.py
│   ├── __init__.py
│   └── receive.py
└── server
    ├── data
    │   ├── __init__.py
    │   ├── interface.py
    │   ├── model.py
    │   └── utilities.py
    ├── __init__.py
    ├── message.py
    └── server.py

4 directories, 15 files
[jon@Adam lxf-example]$ 
```

› **By splitting your code up in to smaller chunks, each placed in its own file and directory, you can bring order to your projects and make future maintenance easier.**

you install Python; it then searches through a list of directories known as the path.

This path is much like the *Bash* shell's **$PATH** environment variable – it uses the same syntax, and serves exactly the same function. It varies, however, in how the contents of the Python path are generated. Initially, the locations stored in the path consist of the following two locations:

- The directory containing the script doing the importing.
- The **PYTHONPATH**, which is a set of directories pre-defined in your default installation.

You can inspect the path in your Python environment by importing the **sys** module, and then inspecting the path attribute (typing **sys.path** will do the trick).

Once a program has started, it can even modify the path itself and add other locations to it.

Variable scope in modules

Before you head off and start merrily writing your own modules, there is one more thing that you need to know about: variable scope.

We have touched upon scope as a concept before, but as a quick refresher, scope refers to the part of a program from which particular variables can be accessed. For instance, a single Python module might contain the following code:

```
food = ["apples", "oranges", "pears"]
print food
def show_choc():
   food = ["snickers", "kitkat", "dairy milk"]
   print food
show_choc()
print food
```

If you run that, you'll see that outside the function the variable **food** refers to a list of fruit, while inside the function, it refers to a list of chocolate bars. This demonstrates two different scopes: the global scope of the current module, in which food refers to a list of fruit, and the local scope of the function, in which food refers to a list of chocolate.

When looking up a variable, Python starts with the innermost variable and works its way out, starting with the immediately enclosing function, and then any functions enclosing that, and then the module's global scope, and then finally it will look at all the built-in names.

In simple, single-file programs, it is a bad idea to put variables in the global scope. It can cause confusion and subtle errors elsewhere in your program. Using modules can help with this problem, because each module has its own global scope. As we saw above, when we import a module, its contents are all stored as attributes of the module's name, accessed via dot notation. This makes global variables somewhat less troublesome, although you should still be careful when using them. ■

Python style

While many people think of Python as a modern language, it's actually been around since the early 1990s. As with any programming language that's been around for any length of time, people who use it often have learned a lot about the best ways to do things in the language – in terms of the easiest way to solve common problems, and the best ways to format your code to make sure it's readable for co-workers and anyone else working on the code with you (including your future self).

If you're interested in finding out more about these best practices in Python, there are two very useful resources from which you can start learning:

- **http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html**
- **www.python.org/dev/peps/pep-0008**

Read these and you're sure to gain some deeper insight into the language.

Embrace storage and persistence

Deal with persistent data and store your files in Python to make your programs more permanent and less transient.

Storage is cheap: you can buy a 500GB external hard drive for less than £40 these days, while even smartphones come with at least 8GB of storage, and many are easily expandable up to 64GB for only the price of a pair of jeans.

It's no surprise, then, that almost every modern application stores data in one way or another, whether that's configuration data, cached data to speed up future use, saved games, to-do lists or photos. The list goes on and on.

With this in mind, this programming tutorial is going to demonstrate how to deal with persistent data in our language of choice – Python.

The most obvious form of persistent storage that you can take advantage of in Python is file storage. Support for it is included in the standard library, and you don't even have to import any modules to take advantage of it.

To open a file in the current working directory (that is, wherever you ran the Python script from, or wherever you were when you launched the interactive shell), use the **open()** function:

```
file = open(“lxf-test.txt”, “w”)
```

The first argument to open is the filename, while the second specifies which mode the file should be opened in – in this case, it's write, but other valid options include read-only (**r**) and append (**a**).

> “Almost every modern application stores data in one way or another”

In previous tutorials, we've shown you that this file object is in fact an iterator, which means you can use the **in** keyword to loop through each line in the file and deal with its contents, one line at a time. Before reviewing that information, however, let's look at how to write data to a file.

Writing to files

Suppose you're writing your own RSS application to deal with your favourite feeds. You've already got some way to ask users to enter in a list of feeds (perhaps using **raw_input()**, or perhaps using a web form and CGI), but now you want to store that list of feeds on disk so you can re-use it later when you're checking for new updates. At the moment, the feeds are just stored in a Python list:

```
feeds = [“http://newsrss.bbc.co.uk/rss/newsonline_uk_
edition/front_page/rss.xml”, “http://www.tuxradar.com/rss”]
```

To get the feeds into the file is a simple process. Just use the **write** method:

```
for feed in feeds:
        file.write(“{0}\n”.format(feed))
```

Easy! Notice how we used the **format string** function to add a new line to the end of each string, otherwise we would end up with everything on one line – which would have made it harder to use later.

Re-using the contents of this file would be just as simple. Using the file as an iterator, load each line in turn into a list, stripping off the trailing new line character. We'll leave you to figure this one out.

When using files in your Python code, there are two things that you need to keep in mind. The first is that you need to convert whatever you want to write to the file to a string first. This is easy, though, because you can just use the built-in **str()** function – for example, **str(42) => “42”**.

The second is that you have to close the file after you have finished using it – if you don't do this, you risk losing data that you thought had been committed to disk, but that had not yet been flushed. You can do this manually with the **close** method of file objects. In our example, this would translate to adding **file.close()** to our program. It's better, however, to use the **with** keyword:

```
with open(“lxf-test.txt”, “a”) as file:
        feeds = [line.rstrip(“\n”) for line in f]
```

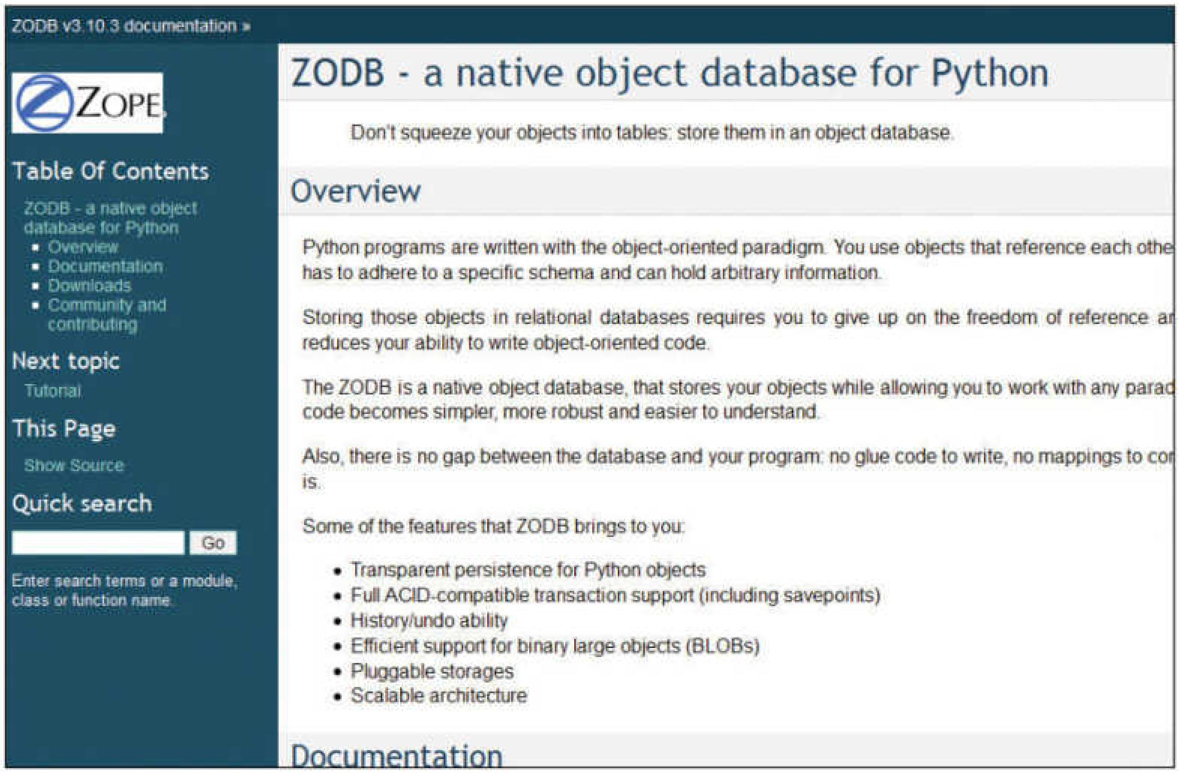
This simple piece of Python code handles opening the file object and, when the block inside the **with** statement is finished, automatically closes it for us, as well. If you are unsure what the second line does, try looking up Python list comprehensions – they're a great way to write efficient, concise code, and to bring a modicum of functional style into your work.

Serialising

Working with files would be much easier if you didn't have to worry about converting your list (or dictionary, for that matter) into a string first of all – for dictionaries in particular, this could get messy. Fortunately, Python provides two tools to make this easier for us.

The first of these tools is the **pickle** module. **Pickle** accepts many different kinds of Python objects, and can then convert them to a character string and back again. You still have to do the file opening and closing, but you no longer have to worry about figuring out an appropriate string representation for your data:

```
import pickle
...
with open(“lxf-test.txt”, “a”) as file:
        pickle.dump(feeds, file)
…
with open(“lxf-test.txt”, “r”) as file:
        feeds = pickle.load(file)
```



ZODB v3.10.3 documentation »

Table Of Contents

- ZODB - a native object database for Python
  - Overview
  - Documentation
  - Downloads
  - Community and contributing

Next topic

Tutorial

This Page

Show Source

Quick search

Enter search terms or a module, class or function name.

ZODB - a native object database for Python

Don't squeeze your objects into tables: store them in an object database.

Overview

Python programs are written with the object-oriented paradigm. You use objects that reference each othe has to adhere to a specific schema and can hold arbitrary information.

Storing those objects in relational databases requires you to give up on the freedom of reference ar reduces your ability to write object-oriented code.

The ZODB is a native object database, that stores your objects while allowing you to work with any parad code becomes simpler, more robust and easier to understand.

Also, there is no gap between the database and your program: no glue code to write, no mappings to cor is.

Some of the features that ZODB brings to you:

- Transparent persistence for Python objects
- Full ACID-compatible transaction support (including savepoints)
- History/undo ability
- Efficient support for binary large objects (BLOBs)
- Pluggable storages
- Scalable architecture

Documentation

› If you're interested in persistent data in Python, a good next stopping point is the ZODB object database. It's much easier and more natural in Python than a relational database engine (www.zodb.org).

This is much easier, and it has other applications outside of persisting data in files, too. For example, if you wanted to transfer your feed list across the network, you would first have to make it into a character string, too, which you could do with **pickle**.

The problem with this is that it will only work in Python – that is to say, other programming languages don't support the pickle data format. If you like the concept of pickling (more generically, serialising), there's another option that does have support in other languages, too: JSON.

You may have heard of JSON – it stands for JavaScript Object Notation, and is a way of converting objects into human-readable string representations, which look almost identical to objects found in the JavaScript programming language. It's great, because it's human readable, and also because it's widely supported in many different languages, largely because it has become so popular with fancy web 2.0 applications.

In Python, you use it in exactly the same way as **pickle** – in the above example, replace **pickle** with **json** throughout, and you'll be writing interoperable, serialised code!

Shelves

Of course, some code bases have many different objects that you want to store persistently between runs, and keeping track of many different pickled files can get tricky. There's another Python standard module, however, that uses **pickle** underneath, but makes access to the stored objects more intuitive and convenient: the **shelve** module.

Essentially, a shelf is a persistent dictionary – that is to say, a persistent way to store key-value pairs. The great thing about shelves, however, is that the value can be any Python object that **pickle** can serialise. Let's take a look at how you can use it. Thinking back to our RSS reader application, imagine that as well as the list of feeds to check, you wanted to keep track of how many unread items each feed had, and which item was the last to be read. You might do this with a dictionary, for example:

```
tracker = { "bbc.co.uk":       { "last-read": "foo",
                                   "num-unread": 10, },
            "tuxradar.co.uk":  { "last-read": "bar",
                                   "num-unread": 5, }}

```

You could then store the list of feeds and the tracking details for each in a single file by using the **shelve** module, like so:

```
import shelve
shelf = shelve.open("lxf-test")
shelf["feeds"] = feeds
shelf["tracker"] = tracker
shelf.close()
```

There are a few important things that you should be aware of about the **shelve** module:

» The **shelve** module has its own operations for opening and closing files, so you can't use the standard **open** function.

» To save some data to the shelf, you must first use a standard Python assignment operation to set the value of a particular key to the object you want to save.

» As with files, you must close the shelf object once finished with, otherwise your changes may not be stored.

Accessing data inside the shelf is just as easy. Rather than assigning a key in the shelf dictionary to a value, you assign a value to that stored in the dictionary at a particular key: **feeds = shelf["feeds"]**. If you want to modify the data that was stored in the shelf, modify it in the temporary value you assigned it to, then re-assign that temporary value back to the shelf before closing it again.

That's about all we have space for this tutorial, but keep reading, because we'll discuss one final option for persistent data: relational databases (for example, *MySQL*). ■

Data organisation and queries

Let's use SQL and a relational database to add some structure to our extensive 70s rock collection. You do have one, right?

In the last tutorial, we looked at how to make data persistent in your Python programs. The techniques we looked at were flat-file based, and as useful as they are, they're not exactly industrial scale. As your applications grow more ambitious, as performance becomes more important, or as you try to express more complicated ideas and relationships, you'll need to look towards other technologies, such as an object database or, even, a relational database.

As relational databases are by far the most common tool for asking complex questions about data today, in this coding tutorial we're going to introduce you to the basics of relational databases and the language used to work with them (which is called SQL, or Structured Query Language). With the basics mastered, you'll be able to start integrating relational databases into your code.

> **"Relational databases are used to ask complex questions about data"**

To follow along, make sure you have got *MySQL* or one of its drop-in replacements installed, and can also find a way to get access to the *MySQL* console:

```
mysql -uroot
```

If you've set a password, use the **-p** switch to give that as well as the username. Throughout, we'll be working on a small database to track our music collection.

Relationships

Let's start by thinking about the information we want to store in our music collection. A logical place to start might be thinking about it in terms of the CDs that we own. Each CD is a single album, and each album can be described by lots of other information, or attributes, including the artist who created the album and the tracks that are on it.

We could represent all this data in one large, homogeneous table – like the one below – which is all well and good, but very wasteful. For every track on the same album, we have to duplicate all the information, such as the album name, its running time, the year it was published, and all the information about the artist, too, such as their name and the year they split. As well as being wasteful with storage space, this also makes the data slower to search, harder to interpret and more dangerous to modify later.

Duplicated data

Album	Free At Last	Free At Last
Artist	Free	Free
Track	Little Bit of Love	Travellin' Man
Track time	2:34	3:23
Album time	65:58	65:58
Year	1972	1972
Band split	1973	1973

Relational database

Album name	Free At Last
Running time	65:58
Year	1972
Artist_id	1

Relational databases solve these problems by letting us split the data and store it in a more efficient, useful form. They let us identify separate entities within the database that would benefit from being stored in independent tables.

In our example, we might split information about the album, artist and track into separate tables. We would then only need to have a single entry for the artist Free (storing the name and the year the band split), a single entry for the album *Free At Last* (storing its name, the year it was published and the running time), and a single entry for each track in the database (storing everything else) in each of their respective tables.

All that duplication is gone, but now all the data has been separated, what happens when you want to report all the information about a single track, including the artist who produced it and the album it appeared on? That's where the 'relational' part of a relational database comes in.

Every row within a database table must in some way be unique, either based on a single unique column (for example, a unique name for an artist, or a unique title for an album), or a combination of columns (for example, the album title and year published). These unique columns form what is known as a primary key. Where a natural primary key (a natural set of unique columns) doesn't exist within a table, you can easily add an artificial one in the form of an automatically incrementing integer ID.

We can then add an extra column to each table that references the primary key in another table. For example, consider the table above. Here, rather than giving all the information about the artist in the same table, we've simply specified the unique ID for a row in another table, probably called Artist. When we want to present this album to a user, in conjunction with information about the artist who published it, we can get the information first from this Album table, then retrieve the information about the artist, whose ID is 1, from the Artist table, combining it for presentation.

SQL

That, in a nutshell, is what relational databases are all about. Splitting information into manageable, reusable chunks of data, and describing the relationships between those chunks. To create these tables within the database, to manage the relationships, to insert and query data, most relational databases make use of SQL, and now that you know what a table and a relationship is, we can show you how to use SQL to create and use your own.

After logging into the *MySQL* console, the first thing we need to do is create a database. The database is the top-level storage container for bits of related information, so we need to create it before we can start storing or querying anything else. To do this, you use **create database**:

```
create database lxfmusic;
```

Notice the semi-colon at the end of the command – all SQL statements must end with a semi-colon. Also notice that we've used lower-case letters: SQL is not case-sensitive, and you can issue your commands in whatever case you like.

With the database created, you now need to switch to it. Much as you work within a current working directory on the Linux console, in *MySQL*, many commands you issue are relative to the currently selected database. You can switch databases with the **use** command:

```
use lxfmusic;
```

Now to create some tables:

```
create table Album (
Album_id int auto_increment primary key,
name varchar(100)
);

create table Track (
Track_id int auto_increment primary key,
title varchar(100),
running_time int,
Album_id int
);
```

The most obvious things to note here are that we have issued two commands, separated by semi-colons, and that we have split each command over multiple lines. SQL doesn't care about white space, so you can split your code up however you like, as long as you put the right punctuation in the correct places.

As for the command itself, notice how similar it is to the **create database** statement. We specify the action we want to take, the type of object that we're operating on and then the properties of that object. With the **create database** statement, the only property was the name of the database; with the **create table** statement, we've also got a whole load of extra properties that come inside the parentheses and are separated by commas.

These are known as column definitions, and each comma-separated entry describes one column in the database. First, we give the column a name, then we describe the type of data stored in it (this is necessary in most databases), then we specify any additional properties of that column, such as whether it is part of the primary key.

The **auto_increment** keyword means you don't have to worry about specifying the value of **Track_id** when inserting data, as *MySQL* will ensure that this is an integer that gets incremented for every row in the database, thus forming a primary key. You can find out more about the **create table** statement in the *MySQL* documentation at **http://dev.mysql.com/doc/refman/5.5/en/create-table.html**.

› *MariaDB* is a drop-in replacement for the *MySQL* database, and is quickly finding favour among distros including Mageia, OpenSUSE and even Slackware.

Inserts and queries

Inserting data into the newly created tables isn't any trickier:

```
insert into Album (name) values ("Free at Last");
```

Once again, we specify the action and the object on which we're acting, we then specify the columns which we're inserting into, and finally the values of the data to be put in.

Before we can insert an entry into the Track table, however, we must discover what the ID of the album *Free At Last* is, otherwise we won't be able to link the tables together very easily. To do this, we use the **select** statement:

```
select * from Album where name = "Free At Last";
```

This command says we want to select all columns from the Album table whose name field is equal to *Free At Last*. Pretty self-explanatory really! If we'd only wanted to get the ID field, we could have replaced the asterisk with **Album_id** and it would have taken just that column.

Since that returned a **1** for us (it being the first entry in the database), we can insert into the Track table as follows:

```
insert into Track (title, running_time, Album_id) values
('Little Bit of Love', 154, 1);
```

The big thing to note is that we specified the running time in seconds and stored it as an integer. With most databases, you must always specify a data type for your columns, and sometimes this means that you need to represent your data in a different manner than in your application, and you will need to write some code to convert it for display. That said, *MySQL* does have a wide range of data types, so many eventualities are covered.

That's all we have space for here, but don't let your *MySQL* education stop there. Now you've seen the basics, you'll want to investigate foreign keys and joins, two more advanced techniques that will enable you to be far more expressive with your SQL. You'll also want to investigate the different types of relationship, such as one-to-one, one-to-many, many-to-one and many-to-many.

Finally, if you want to integrate *MySQL* with your programming language of choice, look out for an appropriate module, such as the **python-mysql** module for Python. ■

CODING
MADE SIMPLE

Raspberry Pi

Discover how you can develop on the low-cost, micro-PC system

Welcome to Pi

Reveal the Raspbian desktop and partake of the pleasures of Python programming on the Raspberry Pi.

The Pi-tailored Raspbian desktop environment received a thorough overhaul in late 2015, making it an even friendlier place to learn, play or work. The Raspbian team has squeezed a great deal into the 3.5GB install. Most importantly, there's a special edition of *Minecraft*, but there's also *Wolfram Mathematica* (for doing hard sums), *LibreOffice* and the *Epiphany* web browser, as well as PDF and image viewers. We're amply catered for programming-wise, too – there's Scratch (the visual coding language), Sonic Pi (the live coding synth), not to mention the Python programming language.

Compared to, say, Windows 10, Raspbian may appear somewhat spartan. Bear in mind that the Pi (particularly non-Pi 2 models) is considerably less powerful than your average desktop computer, so lightweight is the order of the day. It remains no less functional, and thanks to graphical hardware acceleration, it can play high-definition video without batting an eyelid. Software is managed from the Add/Remove Software application. This downloads packages from Raspbian's repositories, which you can trust. It's possible to do this from the command line too. In fact, it's possible to do pretty much anything from the command line.

> **"The Pi is far less powerful than your average desktop computer, but remains no less functional."**

There's a very brief introduction to the terminal in the Mint tutorial *(see page 16)*, which applies equally well here, so feel free to peruse that to learn about commands such as `ls`, `cd` and `mkdir`. It's a good idea to keep the software on your Pi up to date, so let's see how that works from the terminal. We're going to use the *apt-get* package manager, which handles all of the consultations with Raspbian mirrors and does a remarkable job of (un)installing packages and their dependencies with a minimum of fuss. Open LXTerminal from either the menu or the launch bar, and enter the following command:

```
$ sudo apt-get update
```

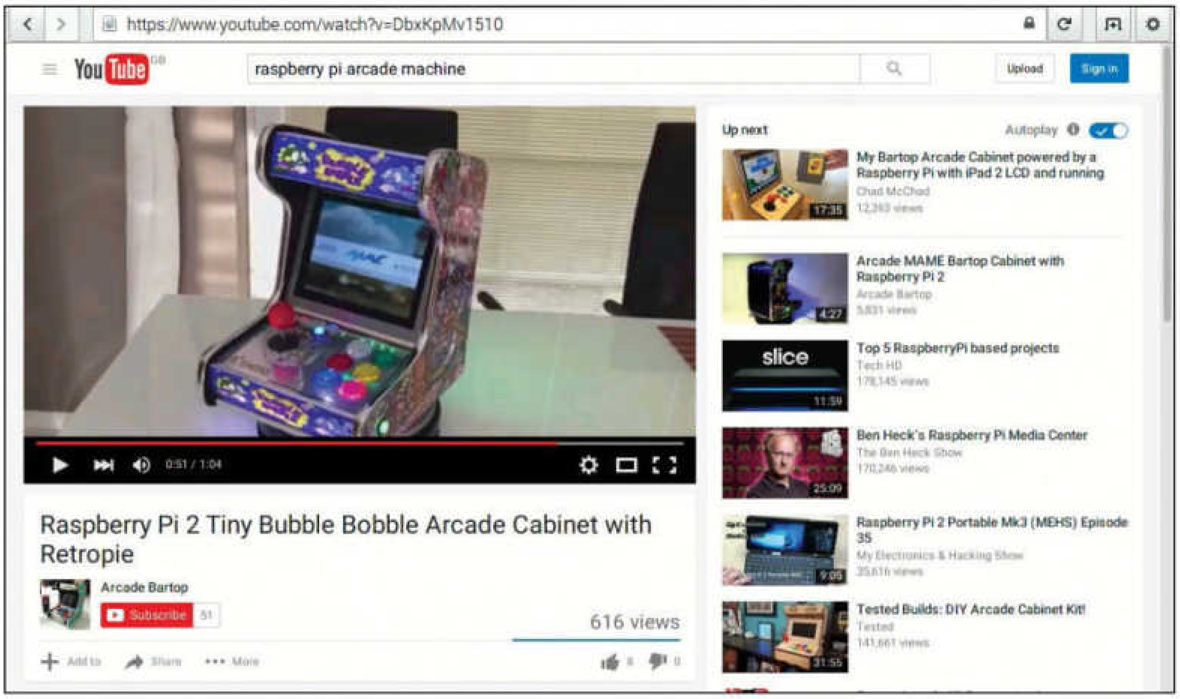
Terminal velocity

That updates Raspbian's local list of all available packages. In the next step, the list of current packages installed on the system is checked against this, and where newer packages are available, these are offered for upgrade:

```
$ sudo apt-get
upgrade
```

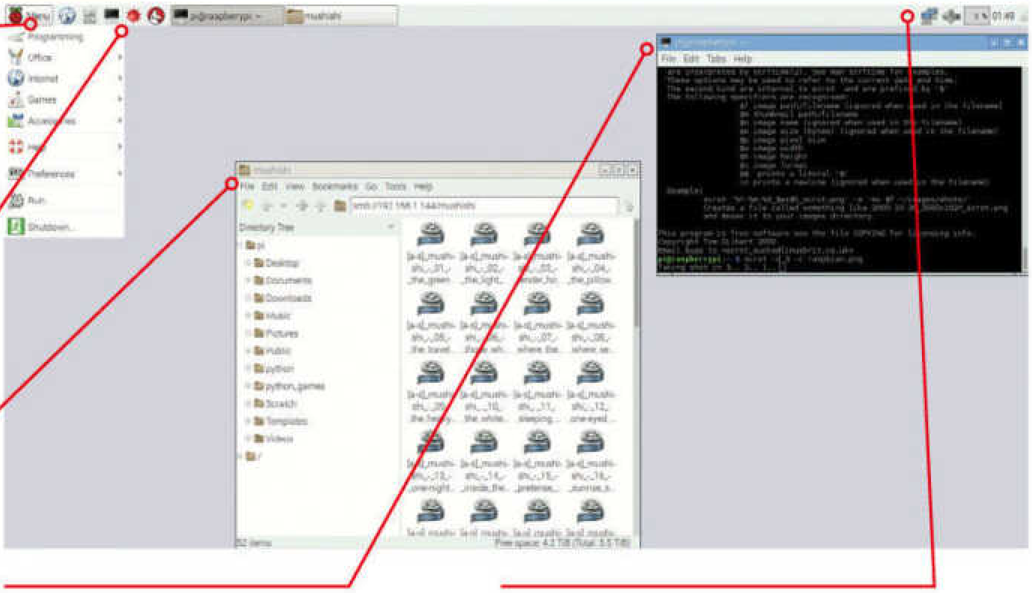
If there's lots of packages to upgrade, the process may take some time. The bottleneck is mostly due to the speed at which data can be written to the SD card – you may want to make a cup of tea. You should update your packages pretty regularly; you'll get important security updates and the latest features or fixes for all your software.

Python is a particularly good first programming language. The syntax is clean and concise, there are no awkward


https://www.youtube.com/watch?v=DbxKpMv1510
YouTube GB
raspberry pi arcade machine
Upload
Sign in
0:51 / 1:04
Raspberry Pi 2 Tiny Bubble Bobble Arcade Cabinet with Retropie
Arcade Bartop
Subscribe 51
616 views
Add to
Share
More
Up next
Autoplay
My Bartop Arcade Cabinet powered by a Raspberry Pi with iPad 2 LCD and running
Chad McChad
12,393 views
17:35
Arcade MAME Bartop Cabinet with Raspberry Pi 2
Arcade Bartop
5,831 views
4:27
Top 5 RaspberryPi based projects
Tech HD
178,145 views
slice
11:59
Ben Heck's Raspberry Pi Media Center
The Ben Heck Show
170,246 views
25:09
Raspberry Pi 2 Portable Mk3 (MEHS) Episode 35
My Electronics & Hacking Show
35,616 views
9:05
Tested Builds: DIY Arcade Cabinet Kit!
Tested
141,661 views
31:55

› **The new *Epiphany* browser works well; it can even play HTML5 YouTube videos, such as this one featuring a tiny arcade cabinet.**

The Raspbian desktop



Menu

Here you'll find all the great programs mentioned earlier – and more. You'll also find links to help resources and, in the Preferences menu, some tools for adjusting system settings.

Launch bar

Frequently used applications can be added here – just right-click it and select Application Launch Bar Settings. The black screen icon in the centre opens the terminal.

File manager

The *PCManFM* file manager is lightweight but thoroughly capable. It opens at your home folder and is also capable of browsing Windows and other network shares.

Terminal

The command line gives direct access to the operating system. It can be daunting but, with practice, you'll appreciate how efficient it is.

Network

From here you can configure your wired and wireless network connections. Some wireless adapters work better with the Pi than others, so it's worth doing some research before you buy.

semicolons to terminate statements, lots of shortcuts for things that are cumbersome in other languages and there's a huge number of modules that enable you to achieve just about anything.

A tale of two Pythons

Raspbian comes with two versions of Python installed. This reflects a duality that has formed in Pythonic circles over the last few years. Ultimately, Python 3 (*see page 94 and 120*) has been around since 2008, but people have been slow to adopt. Many of the larger projects that used to only work on the old version have been ported to the new, so finally the shift to 3 is gaining traction. Sooner or later, you'll run into something that requires Python 2.7, and then you might have to unlearn some things, but for this tutorial we're going to stick with the new version of Python provided by Raspbian – 3.4.

Open up a terminal, if you haven't already done so, and start the Python 3 interpreter by entering:

```
$ python3
```

Once again, if you have a look at the introductory Mint tutorial *(see page 16)*, you'll learn about a few things you can do here. For example, if you wanted to find out how many seconds there are in a year, you could do the following:

```
>>> 60 * 60 * 24 * 365
```

As mentioned elsewhere, it makes more sense to save your programs in text files. So we'll now leave the interpreter by hitting Ctrl+D, and make our first Pi Python program. At this point, you have a choice: you can proceed as in the Mint tutorial and use the *nano* text editor, or you can use the graphical one located in the Accessories menu. Before you make that choice, though, we'll create a directory for storing our Python programs. Keeping on top of your filing is A Good Thing, too.

```
$ mkdir ~/python
```

Now type the following into your chosen editor

```
print('Hello World')
```

and then save the file. Assuming you're using the graphical editor, choose, File > Save As, and then navigate to the recently created Python folder, and save it as **helloworld.py**. If you're using *nano,* use Ctrl+X to save and exit.

Now we can run this and see what we have created:

```
$ cd ~/python
$ python3 helloworld.py
```

It's not going to win any awards, but it's a fully working program and something you can build around in further programming adventures. ■

Other programming tools

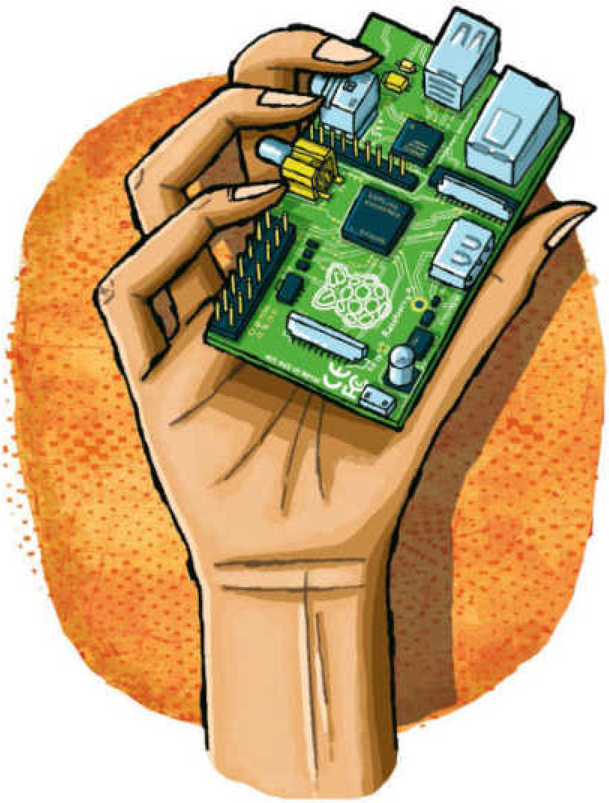
We've only covered Python in this tutorial but there are other programming languages available in Raspbian. To start with, there's a full-blown Java IDE called *Greenfoot* (which has been developed in partnership with the University of Kent and Oracle). This teaches the basics of open-source design using visual scenarios but you can also play with the underlying Java source (if you're feeling brave).

Then there's also Node-RED, for designing applications based around the Internet of Things, for which the Raspberry Pi is well suited. It's easy to add sensors, cameras or LEDs, and turn your Pi into a bona fide smart device. It can watch your letterbox, monitor the weather, or even water your plants.

Finally, Scratch (*see page 84*) is a great way to get kids (or grown-up kids) into coding. It's a visual language, which means that programs are created by dragging and connecting blocks that represent events and logic. Plus it has an orange cat as its mascot. Scratch was developed at MIT and, to date, some 12.5 million projects have been shared on its website at **https://scratch.mit.edu**. Fortune has it that by turning to the very next page, you will find an entire tutorial devoted to the language.

Starting Scratch

Discover how to build a cat and mouse game using this straightforward beginner's programming language.



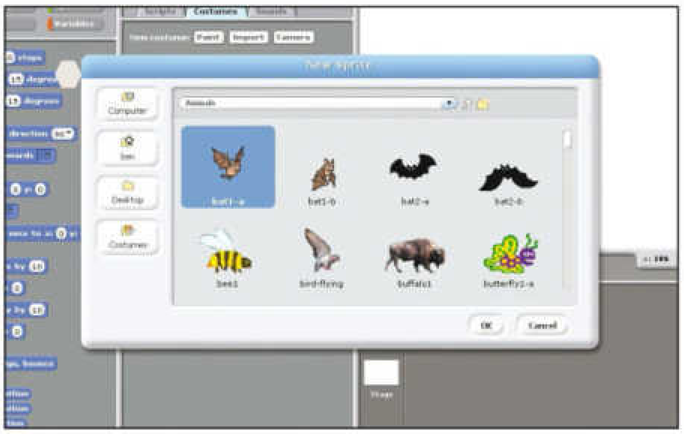
You can use a wide range of programming languages with the Raspberry Pi, and the little learning computer is our focus in this series of articles, although Scratch can be downloaded from **https://scratch.mit.edu** whether you're using Windows, Mac or Linux. These tutorials will be equally applicable whatever your platform.

It's a great beginner's language, because it introduces many of the concepts of programming while at the same time being easy to use. It's especially good for creating graphical programs such as games. If you've never programmed before, or if you're new to graphical programming, we'll ease you gently in, and hopefully have a little fun along the way. Without further ado, let's get started.

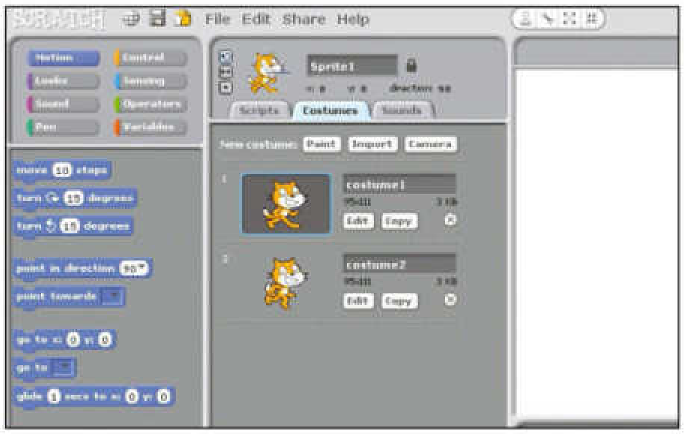
You'll find Scratch on the desktop in core Raspberry Pi operating system Raspbian, so there's no need to install anything – just click on the icon to get started. It's best to use Raspbian rather than one of the other distros for the Pi for this, because not many of the others support Scratch.

The main window is split up into sections. Broadly speaking, the bits you can use to make your programs are on the left, while you make your programs in the middle, and the programs run on the right. Each program is made up of a number of sprites (pictures) that contain a number of scripts. It's these scripts that control what happens when the program is running.

In the top-left corner, you'll see eight words: Motion, Looks, Sound, Pen, Control, Sensing, Operations and Variables. Each of these is a category that contains pieces that you can drag and drop into the scripts area to build programs. Good luck!



❯ Scratch comes with a range of images that you can use for sprites. Click on 'New Sprite From File', or 'Costumes > Import' to see them.



❯ Each sprite can have a number of costumes, or images. Click on the Costumes tab to create new ones or manage existing ones.

Variables and messages

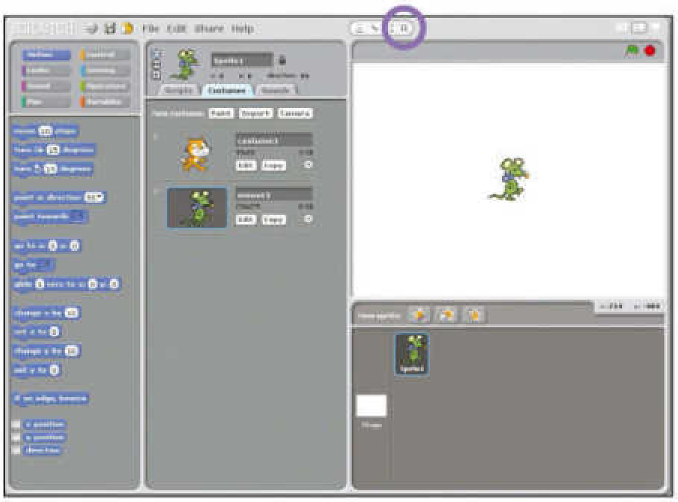
Sooner or later, you're going to want to get your program to remember something. It might be a number, a piece of text, or anything. You can do this using variables. These are little pieces of the computer's memory that your program can place pieces of data in. In step 3, we create a pair of these to store some numbers in, although we could also put text in them. Note that in some programming languages, you have to create different types of variables if you want to store different types of data, but you don't need to worry about that in Scratch.

Once you have created a variable, you can then use it in a few ways. Firstly, you have to set them to be a particular value, then you can use them to evaluate conditions (which we'll do in steps 6 and 11), or you can output them.

Messages

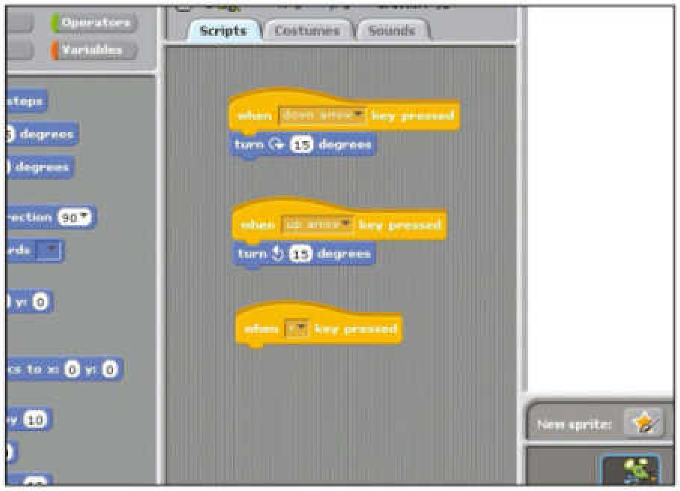
If you create a number of scripts, you may need to communicate between them. Sometimes you can do this with variables, but it is often better to use messages. These can be used to trigger scripts in the same way as keypresses can. When one script broadcasts a message, it will then trigger all the scripts that start with **When I Receive ...** Like variables, messages have names, so they have to be created first, and for a script to trigger it has to be linked to the same message as the broadcast.

Step by step



1 Create the mouse

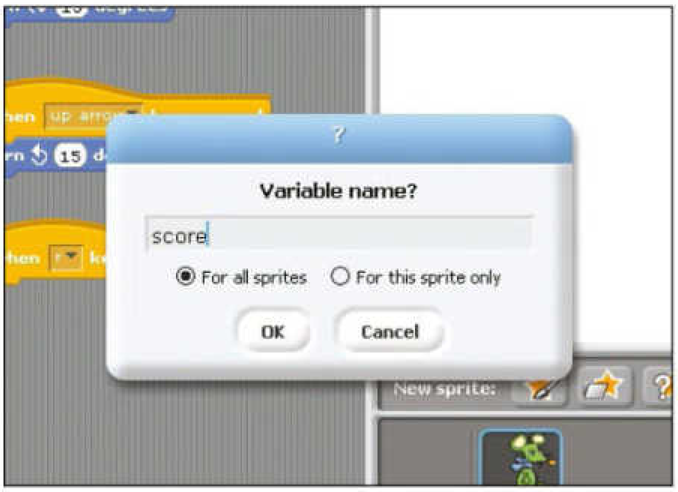
Change the image from a cat to a mouse by going to 'Costumes > Import > Animals > Mouse 1'. Then reduce the sprite size by clicking on the 'Shrink sprite' icon (circled) and then the mouse. We set it to about the size of our thumbnail.



```
when [down arrow] key pressed
turn ↻ 15 degrees

when [up arrow] key pressed
turn ↺ 15 degrees

when [ ] key pressed
```

2 Set keys

Click on 'Scripts', and change **When Right Arrow Key Pressed** to **When r Key Pressed**. We'll use this to start a new game (**r** is for reset). Then drag **Move 10 Steps** off the bottom of the script. If you drop it back in the left side, it will be deleted.



?

Variable name?

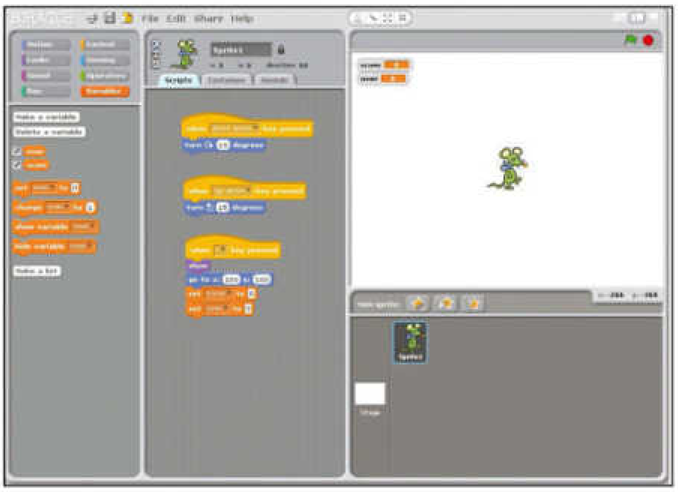
score

● For all sprites ○ For this sprite only

OK Cancel

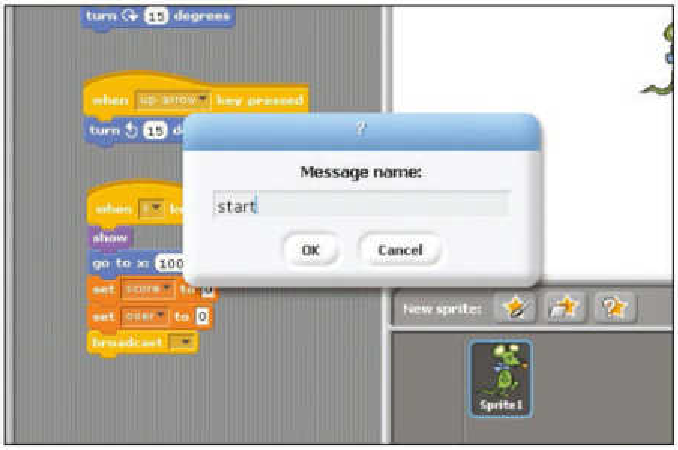
3 Create and name variable

Click on 'Variables' in the top-left (see boxout above for more details on what they are). Click on 'Make A Variable' and enter the variable name as **score**. Repeat the process to create a variable called **over**.



4 Reset the score

Under the script **When r Key Presses**, add the lines **show** (from 'Looks'), **Go To X:100**, **Y:100** (from 'Motion', don't forget to change 0s to 100s), **Set Score To 0** and **Set Over to 0** (both from 'Variables').



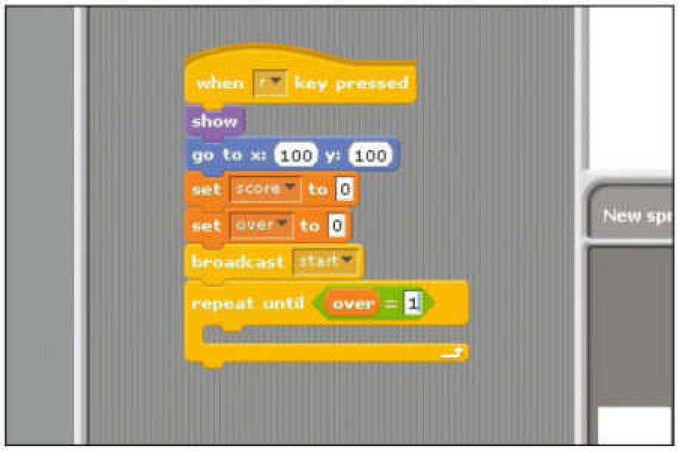
```
turn ↻ 15 degrees

when up arrow key pressed
turn ↺ 15 d

when r key pressed
show
go to x: 100
set score to 0
set over to 0
broadcast
```

Message name:

start

5 Add broadcast

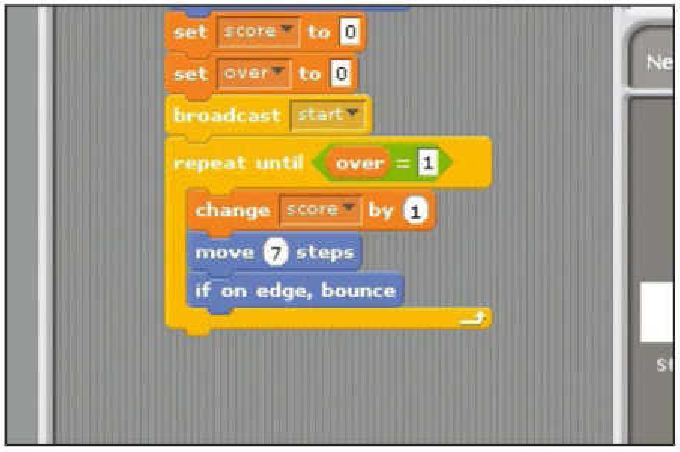
Add the block **Broadcast ...** to the bottom of the **When r Key Pressed** script. Once it's there, click on the drop-down menu and select 'New..'. and give the message the name **start**. We'll use this to let the other sprite know that the game has started.



```
when r key pressed
show
go to x: 100 y: 100
set score to 0
set over to 0
broadcast start
repeat until over = 1
```

6 Create a loop

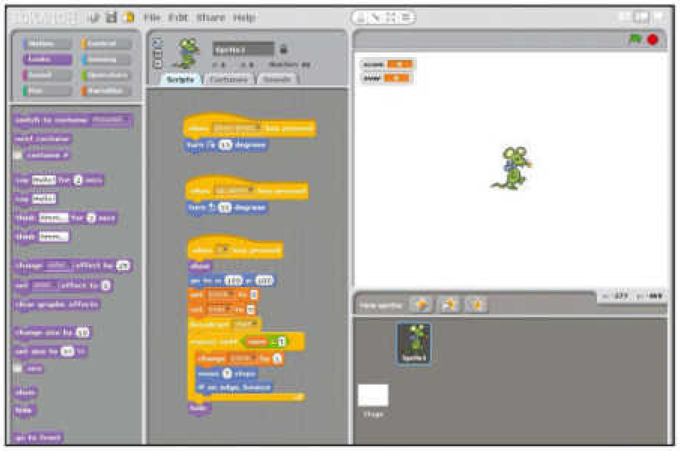
We can create loops that cycle through the same code many times. Continue the script with **Repeat Until ...** (from 'Control'), and then drag and drop **... = ...** (from 'Operators'), then drag **Over** (from 'Variables') into the left-hand side of the **=** and enter **1** on the right.



```
set score to 0
set over to 0
broadcast start
repeat until over = 1
    change score by 1
    move 7 steps
    if on edge, bounce
```

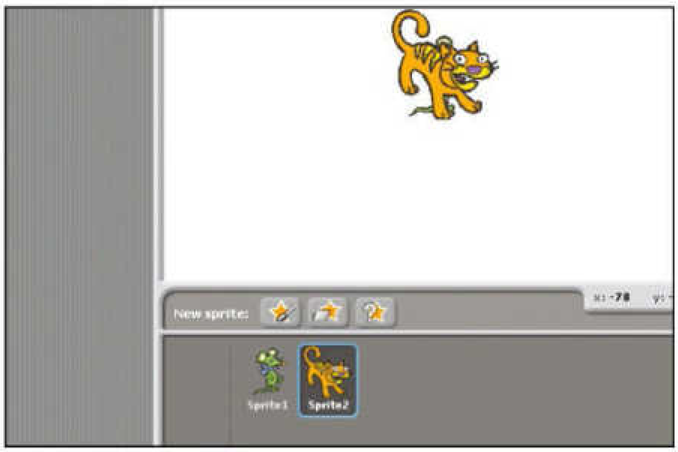
7 Add to your loop

Inside the **Repeat Until Over = 1** block, add **Change score By 1** (from 'Variables'), **Move 7 Steps** (from 'Motion') and **If On Edge, Bounce** (also from 'Motion'). These three pieces of code will be constantly repeated until the variable **over** gets set to **1**.



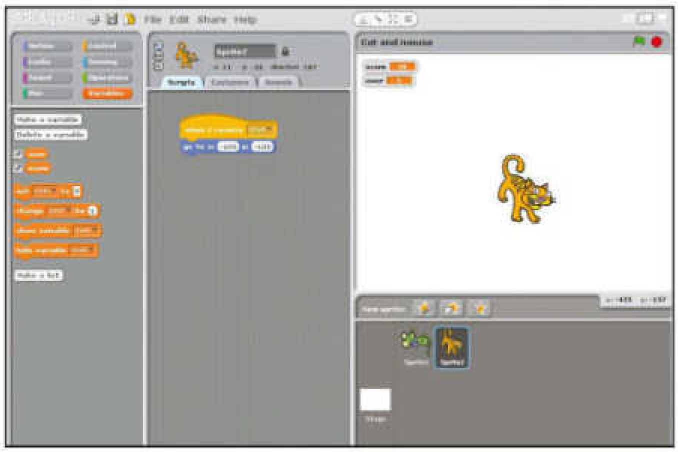
8 Hide the mouse

Once the game has finished (and the cat has got the mouse), the **Repeat Until** loop will end and the program will continue underneath it. Drag **Hide** (from 'Looks') under the loop, so the mouse disappears when this happens.



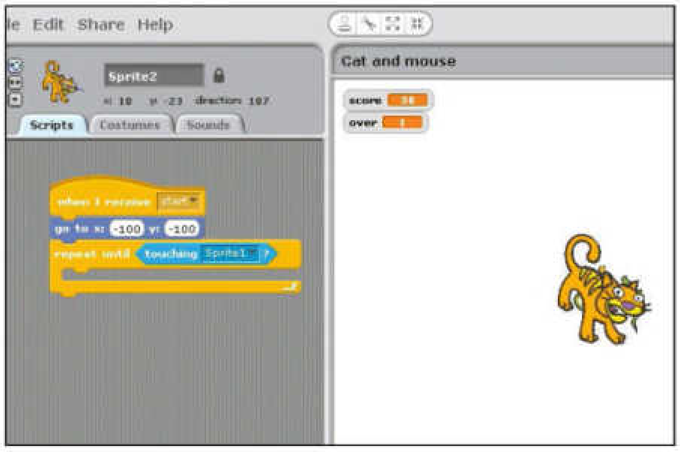
9 Resize your cat

Select 'Choose New Sprite From File > Cat 4', and shrink the sprite down to an appropriate size, as we did with the mouse. Each sprite has its own set of scripts. You can swap between them by clicking on the appropriate icon in the bottom-right.

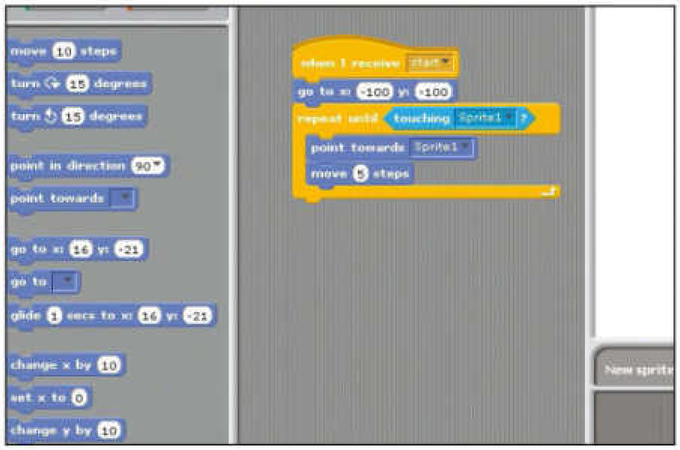


10 Move the cat

In the scripts for the new sprite, start a new script with **When I Receive start** (from 'Control'), and **Go To X:-100 Y:-100**. This will move the cat over to the opposite corner of the screen from the mouse. (0,0) is the middle.



```
when I receive start
go to x: -100 y: -100
repeat until touching Sprite1 ?
```



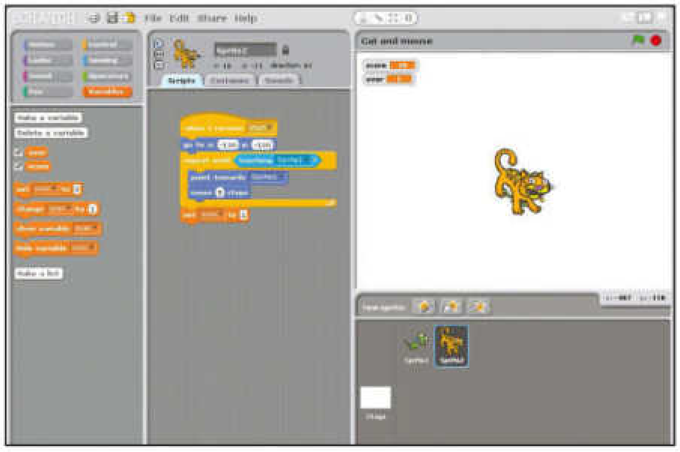
```
when I receive start
go to x: -100 y: -100
repeat until touching Sprite1 ?
    point towards Sprite1
    move 5 steps
```

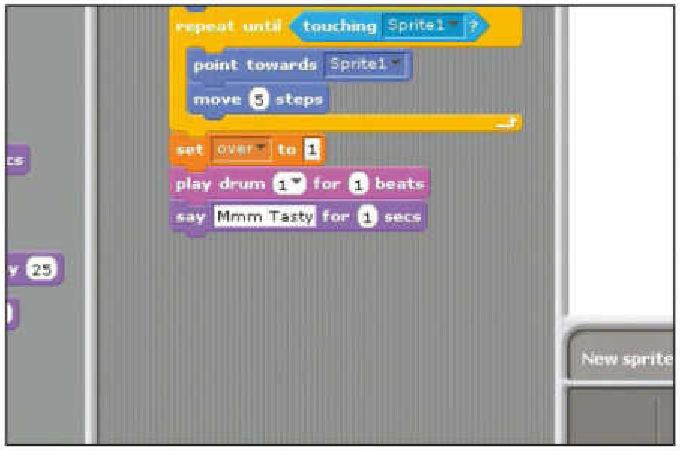
11 Give the cat a loop

As with the mouse, the cat also needs a loop to keep things going. Add **Repeat Until** (from 'Control'), and then in the blank space add **Touching Sprite 1** (from 'Sensing'). This will keep running until the cat (sprite 2) catches the mouse (sprite 1).

12 Set the difficulty

Inside the **Repeat Until** block, add **Point Towards Sprite 1** (from 'Motion') and **Move 4 Steps** (also from 'Motion'). The amount the cat and mouse move in each loop affects the difficulty of the game. We found 4 and 7, respectively, to work well.





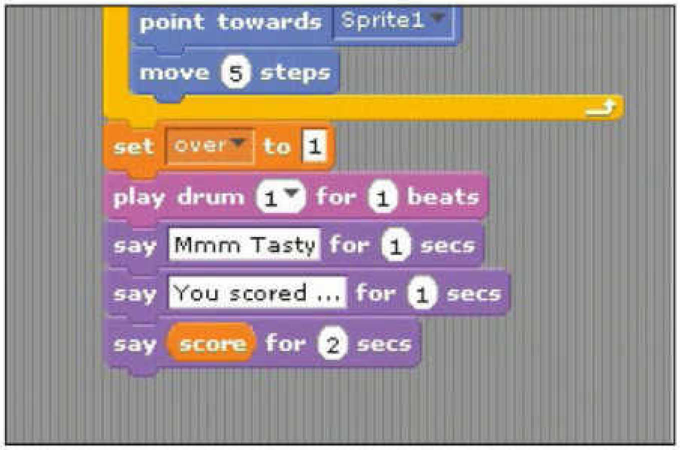
```
repeat until touching Sprite1 ?
    point towards Sprite1
    move 5 steps
set over to 1
play drum 1 for 1 beats
say Mmm Tasty for 1 secs
```

13 Finish the loop

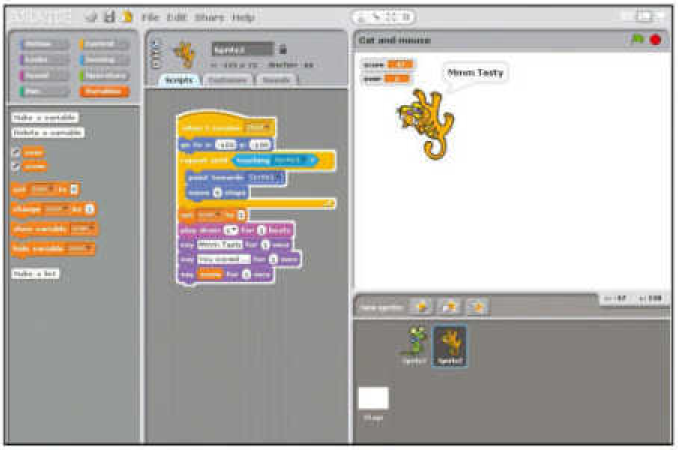
The loop will finish when the cat has caught the mouse – the game is over, so we need to stop the script on Sprite 1. We do this by adding **Set over To 1** (from 'Variables') underneath the **Repeat Until** block. This will cause Sprite 1's main loop to finish.

14 Tell the player the game is over

We now want to let the player know that the game is over. We will do this in two ways: with audio, and on screen. Add **Play Drum 1 for 1 Beats** (from 'Sound'), then **Say Mmmm Tasty For 1 Secs** (from 'Looks').



```
    point towards Sprite1
    move 5 steps
set over to 1
play drum 1 for 1 beats
say Mmm Tasty for 1 secs
say You scored ... for 1 secs
say score for 2 secs
```



15 Display a score

Finally, we can let the player know their score. We increased the variable score by one every loop, so this will have continued to go up. Add **Say You Scored ... For 1 Secs**, then drag another **Say ... for 1 Secs** block and then drag **score** (from 'Variables') into the blank field.

16 Play your game!

Press [r] and play! You can use the up and down arrows to move the mouse around. You can make it easier or more difficult by changing the size of the sprites and the amount they move each loop. Good luck and happy gaming! ■

Further Scratch

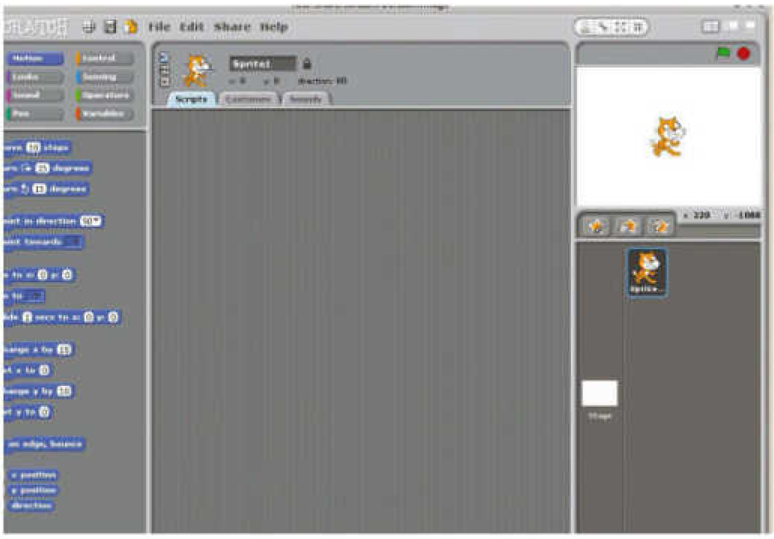
We've dived in, but let's now look at the fundamentals of Scratch to understand how our programs work and set up a straightforward quiz.

How did you learn to program? Typically, we think of a person sitting in front of a glowing screen, fingers slowly typing in magic words that, initially, mean nothing to the typist. And in the early days of coding, that was the typical scene, with enthusiasts learning by rote, typing in reams of code printed in magazines.

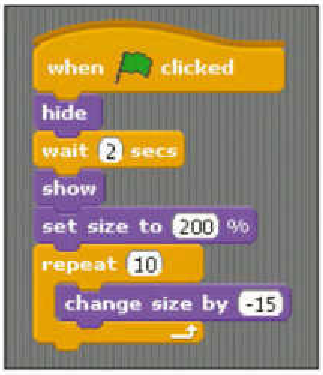
In this modern era, when children are now encouraged to learn coding concepts as part of their primary school education, we see new tools being used to introduce coding to a younger generation, and the most popular tool is the subject of this very tutorial.

Scratch, created by MIT, is a visual programming environment that promotes the use of coloured blocks over chunks of code. Each set of blocks provides different functionality and introduces the concepts of coding in a gentle and fun way. Children as young as six are able to use Scratch, and it's now being heavily used in the UK as part of the curriculum for Key Stages 1 to 3 (6 to 14 years old), and as part of the Code Club scheme of work.

For the purpose of this tutorial, we are using the current stable version of Scratch, which at the time of writing is 1.4. Version 2 of Scratch is available as a beta and reports suggest that it's very stable to use, but there's a number of differences between the locations of blocks when comparing version 1.4 to version 2.

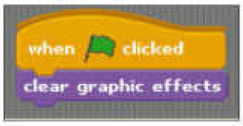


Scratch uses a three-column UI. From left to right: Block Palette, Script Area and The Stage.



```
when [green flag] clicked
hide
wait 2 secs
show
set size to 200 %
repeat 10
    change size by -15
```

Fig 1.0 The logic for our logo sprite.



```
when [green flag] clicked
clear graphic effects
```

Fig 1.2 This is the code that clears effects.

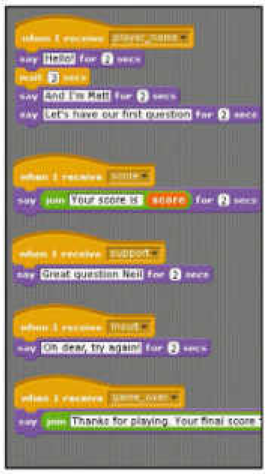
As mentioned, Scratch uses a block-based system to teach users how different functions work in a program. This can be broken down into the following groups and colours:

- **Motion (dark blue)** This enables you to move and control sprites in your game.
- **Control (orange)** These blocks contain the logic to control your program (loops and statements) and the events needed to trigger actions, such as pressing a key. In Scratch 2.0, the events stored in Control have their own group, which is called, naturally enough, Events.
- **Looks (purple)** These blocks can alter the colour, size or costume of a sprite, and introduce interactive elements, such as speech bubbles.
- **Sensing (light blue)** For sensing handles, the general input needed for your program, for example, keystrokes, sprite collision detection and the position of a sprite on the screen.
- **Sound (light purple)** Adds both music and sound effects to your program.
- **Operators (green)** This enables you to use mathematical logic in your program, such as Booleans, conditionals and random numbers.
- **Pen (dark green)** This is for drawing on the screen in much the same way that logo or turtle enable you to do so.
- **Variables (dark orange)** Creates and manipulates containers that can store data in your program.

By breaking the language down into colour-coded blocks Scratch enables anyone to quickly identify the block they

Scratch online

Scratch is available across many platforms. It comes pre-loaded on every Raspberry Pi running Raspbian and is available for download from **http://scratch.mit.edu**. But did you know that there is an online version that provides all of the functionality present in the desktop version, but requires no installation or download? Head over to the MIT website (above) and you'll be greeted with the same interface and blocks of code, but with a few subtle differences. This is the latest version of Scratch and you can save all of your work in the cloud ready to be used on any PC you come across. If you wish to use the code built online on a PC that does not have an internet connection, you can download your project and run it offline. You can also download the latest version of Scratch from the website. This very latest version is still in beta but reports show that it is very stable and ready for use. Scratch 2.0 uses *Adobe Air* and is available across all platforms including Linux. Adobe dropped its support for *Air* on Linux a few years ago but you can still download the packages necessary for Scratch 2.0 to work.



```
when I receive player_name
say Hello! for 2 secs
wait 3 secs
say And I'm Matt for 2 secs
say Let's have our first question for 2 secs

when I receive score
say join Your score is score for 2 secs

when I receive support
say Great question Neil for 2 secs

when I receive insult
say Oh dear, try again! for 2 secs

when I receive game_over
say join Thanks for playing. Your final score
```

Fig 1.1 The code used for Matt to receive broadcasts.

need. Children typically work via the colour-coding system at first, and then through the natural process of playing they understand the link between each of the blocks and how they work together.

The environment

Scratch uses a clear and structured layout, which is divided into three columns. The first column is:

Block palette This is where our blocks of code are stored and sorted by function.

Script area In the second column is an area where you can drag our blocks of code from the block palette to add code to our program.

The stage The third and final column shows the results of your programming and can be used to interact with the game world. At the bottom of this column you will also find the very handy Sprites pane. This shows the sprites and assets that belong to your particular program. Clicking on a sprite will change the focus to that sprite, enabling you to write code for that sprite only.

Building our game

To write code with Scratch, we need to move blocks of 'code' from the block palette to the script area for each sprite or background that we want to use in our program.

For this tutorial, we are going to make a quiz game, using two quiz masters, called Matt and Neil. The purpose of the game is to score more than three points, and you must answer the question correctly to progress to the next round. If you receive three wrong answers in the game, it's game over. Each of our sprites has their own scripts assigned to run once an event triggers it. An event is typically clicking on the green flag to start the game but it can be as complex as listening for a trigger from another sprite.

We've given our four game sprites names: Neil is Sprite6, Matt is Sprite7, our logo is Sprite5 and the Game Over image is Sprite8. Each of these sprites has their own scripts associated with them, so let's look at what they each do, starting with the logo. We wanted this logo to appear right at the start of the game, but not be visible straight away, so our logic was as follows:

```
When Green Flag clicked.
Hide.
Wait for 2 seconds.
Show.
Double in size.
Loop 10 times and each time reduce size by 15%
```

You can see this as Scratch presents it visually in Fig 1.0.

Now let's look at the logic for Matt's sprite. Matt has five sections of code that react to something called a 'broadcast', which is basically a way to chain trigger events between sprites in the program (you can find 'broadcast' under 'Control'). So we have a number of actions to perform once we receive these broadcasts from Neil and the stage.

player_name The stage sends a broadcast once the player has entered their name.

support Neil sends a broadcast to Matt once the player has answered a question correctly.

insult Neil sends a broadcast to Matt to taunt the player.

Score Neil sends a broadcast to Matt triggering Matt to tell the player their score.

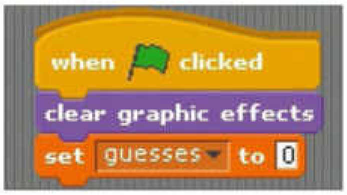
game_over Neil sends Matt a broadcast to trigger the end of the game.

As we can see (Fig 1.1, left), for each of these broadcasts Matt is triggered to run a certain sequence of code. Matt also has a script that's run in the event of clicking on the green flag. Fig 1.2 (see left) shows the code that clears any special effects used on his sprite.

Let's move on to Neil. This sprite has a lot more code than Matt. This is because he's the main part of our game.

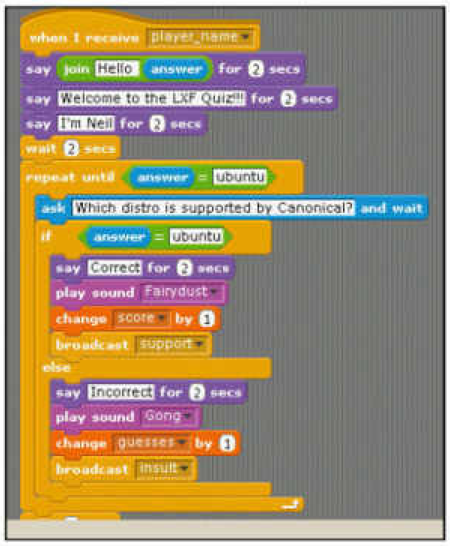
First, there's the Green Flag event. What this does is triggers Neil's sprite to reset any special effects that may be in use and then stores 0 in the variable called guesses (see Fig 1.3, above-right).

The second part is the loop that controls our game. This large loop is triggered once we receive the name of the player, which is handled via the code associated with the stage, which we will come to later.



```
when [flag] clicked
clear graphic effects
set guesses to 0
```

Fig 1.3 This resets Neil's sprite and the guesses variable.



```
when I receive player_name
say join Hello answer for 2 secs
say Welcome to the LXF Quiz!!! for 2 secs
say I'm Neil for 2 secs
wait 2 secs
repeat until answer = ubuntu
  ask Which distro is supported by Canonical? and wait
  if answer = ubuntu
    say Correct for 2 secs
    play sound Fairydust
    change score by 1
    broadcast support
  else
    say Incorrect for 2 secs
    play sound Gong
    change guesses by 1
    broadcast insult
```

Fig 1.4 Part 1 of the main game code.

We've broken Matt's main loop into three parts. At the start of the game, there's a section of code associated with the stage that asks for the player's name. It then stores that as a variable called **Answer**, which is stored in the Sensing category of blocks. Once the user enters their name, the script associated to the stage sends a broadcast called **player_name**. Neil's code is waiting to receive this broadcast as a trigger event. Once the code is triggered, Neil will say hello to the player by their name, using the **Answer** variable.

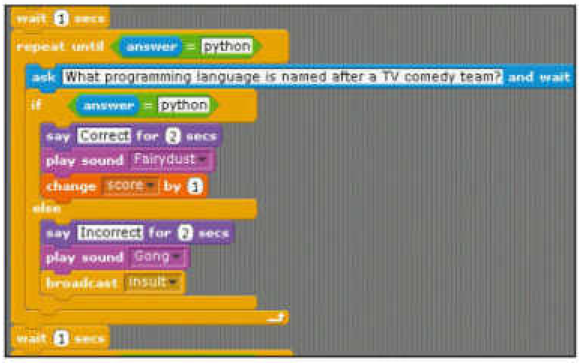
Once we have the formalities out of the way, we move on to the main loop that controls the first question. We wrap the question in a loop that will repeat until the correct answer is given, which is **repeat until answer = ubuntu**.

We then ask the question and run another loop inside of the main loop (see Fig 1.5, below). This second loop is an **if** statement which uses a condition to perform a certain sequence of code if a condition is true, or if it is false it will run the code associated with else. In the case of the answer being

»

Quick tip

Right-clicking on a block of code is a quick way to duplicate or delete your code.



```
wait 1 secs
repeat until answer = python
  ask What programming language is named after a TV comedy team? and wait
  if answer = python
    say Correct for 2 secs
    play sound Fairydust
    change score by 1
  else
    say Incorrect for 2 secs
    play sound Gong
    broadcast insult
wait 1 secs
```

Fig 1.5 The second loop of Neil's code inside the main loop.

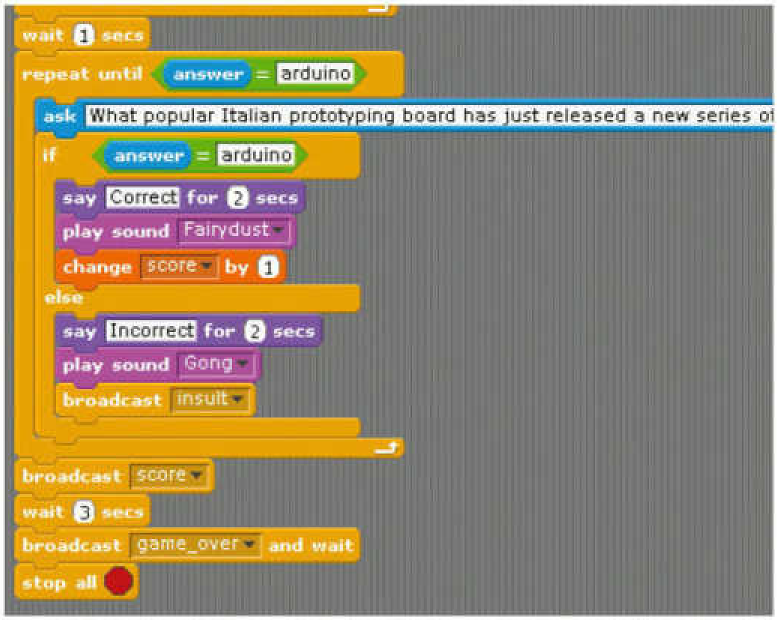
Quick tip

Blocks that can be linked together will have a white 'halo' indicating they can be connected.

correct, Neil will say that the answer is correct, play a sound to reward the player, then alter the variable score by 1 point and, finally, broadcast support to Matt, who will say something nice. If the player provides an incorrect answer, then Neil will say 'Incorrect', play a gong sound effect, and then increment the guesses variable by 1 and send a broadcast to Matt who will taunt the player.

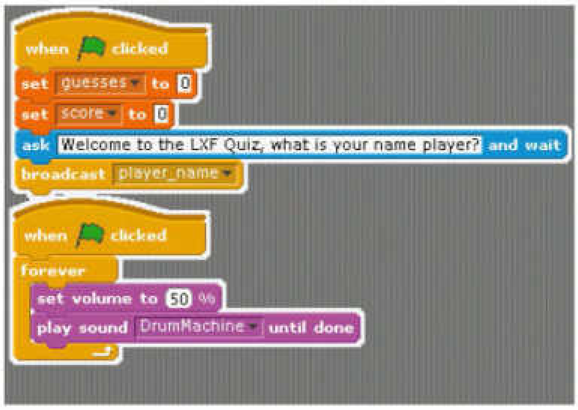
Part two of the code is exactly the same as the main loop of part one and the reason why is because we duplicated the code and changed the question and expected answer. To duplicate code in Scratch, you can simply right-click on the blocks of code and select 'Duplicate'. Hey presto – you have doubled your code in just one click.

The first section of the third part is exactly the same as the main loop from the previous two parts. So let's move down to the last four blocks of code (see Fig 1.6, below). Our first block is a broadcast **score** to Matt – this will trigger Matt to tell us the score. We then pause for three seconds to allow Matt to finish speaking. Then we send another broadcast to Matt, who will then run thorough the end of game code associated with that broadcast. Lastly, we use **stop all** to stop any scripts in the game.



```
wait 1 secs
repeat until answer = arduino
  ask What popular Italian prototyping board has just released a new series of
  if answer = arduino
    say Correct for 2 secs
    play sound Fairydust
    change score by 1
  else
    say Incorrect for 2 secs
    play sound Gong
    broadcast insult
broadcast score
wait 3 secs
broadcast game_over and wait
stop all
```

Fig 1.6 The last part of the main code associated with Neil.



```
when [flag] clicked
set guesses to 0
set score to 0
ask Welcome to the LXF Quiz, what is your name player? and wait
broadcast player_name

when [flag] clicked
forever
  set volume to 50 %
  play sound DrumMachine until done
```

Fig 1.7 The Stage contains the sprites but it can also have its own scripts.

The Green Flag event is the code that controls the number of guesses that the player has. We use conditional logic to say that when the number of guesses is equal to 3, we broadcast **game_over** to Matt, which, in turn, triggers the end of game script.

The Game Over sprite has two scripts associated with it. The first is simply that when the green flag is clicked, to hide the sprite. The second script is triggered when the **game_over** broadcast is sent. It triggers the sprite to reveal itself and set its size to 100%. We then use a loop that will repeat 10 times to change the size and rotation of the sprite, giving us a rotating zooming effect, just like you'd see in the classic 8-bit games of the 1980s.

The stage

As well as being the home of our sprites, the stage can also contain its own scripts (see Fig 1.7, above). For our game we have two sections of code on the stage. Both sections of code are triggered by the click on Green Flag event. The first part resets two variables called **guesses** and **score**. We then ask the player to provide their name, which is then broadcast to Matt and Neil, and starts the main code loop assigned to Neil. The second section of code is an infinite loop that will play the loop **DrumMachine** continuously and set its volume to 50%.

As we mentioned earlier on, variables are a great way to store data. But before we can use one, we need to create it. To create a variable, we need to use the 'Variables' button from the block palette. In there you will find the 'Make a variable button'. Click on it and you will see Fig 1.9 (see right).

In our game we used two variables – **score** and **guesses** – and we want them both to be available for all sprites, so that Matt and Neil can use them both. Once created, we can easily

Programming concepts

Using Scratch is great fun but did you realise that you are also learning to code? No matter what language you use, the underlying concepts of coding provide a firm foundation. And once learnt they can be applied to any coding project. The main concepts are:

» **Sequences** A series of tasks required to be completed in a certain order. For example, the steps needed to solve a maze.

» **Loops** A way to repeat a sequence. They can be run for ever (**while true**) or controlled using a for statement (**for x in range(0,3)**). We have used many loops to control the player's progress in our game.

» **Parallelism** This is the principle of running more than one sequence of code at the same time. We've used that a lot in our Scratch game because each sprite has its own code that runs in parallel to each other.

» **Events** This is a trigger that starts a sequence of code, and the most visible event in our Scratch game here is clicking on the green flag to start the game.

» **Data** We use a variable to store the value of our score and we can later retrieve and manipulate the score to show the player's progress through the game.

» **Operators** These are the basic mathematical rules that we all learn in school. We can apply operators to text and numbers, which enables us to perform calculations in our code and iterate data if required.

» **Conditionals** These form the basis of our logic and provide a method for us to compare data against the input that is given by the player. We have used conditionals in our game to compare the answer given to the expected answer. If they both matched, which in Boolean logic would be classed as True, the player would be awarded a point. If they did not match, which would be defined as False, the player would have to try once again.

drop these variables into our code, enabling us to reuse their value many times in the game, see Fig 1.8 below for the example that we made for our game.

Pseudo code

When trying to understand the logic of our game, we like to write pseudo code. 'Pseudo what?' we hear you say. This is when you write down the logic of how your program will work. Let's look at a simple example:

```
a has the value of 0
while a is less than 10:
print on the screen the value of a
increment a by 1
```

So we have our pseudo code, but how do we express this in a programming language? First, let's do this with Scratch followed by Python. In Scratch, our code will look like this:

```
[When Green Flag is clicked]
Set variable a to 0
forever if a < 10
    say a for 2 secs
    change a by 1
```

This gives the variable **a** the value of 0. We then create a conditional loop that only loops while **a** is less than 10. Inside the loop, we ask Scratch to print the value of **a** for 2 seconds, then increment the value of **a** by 1. This loop will continue until we reach **a = 9** and then it will stop as the next value, 10, is not less than 10. In Python our code looks like this

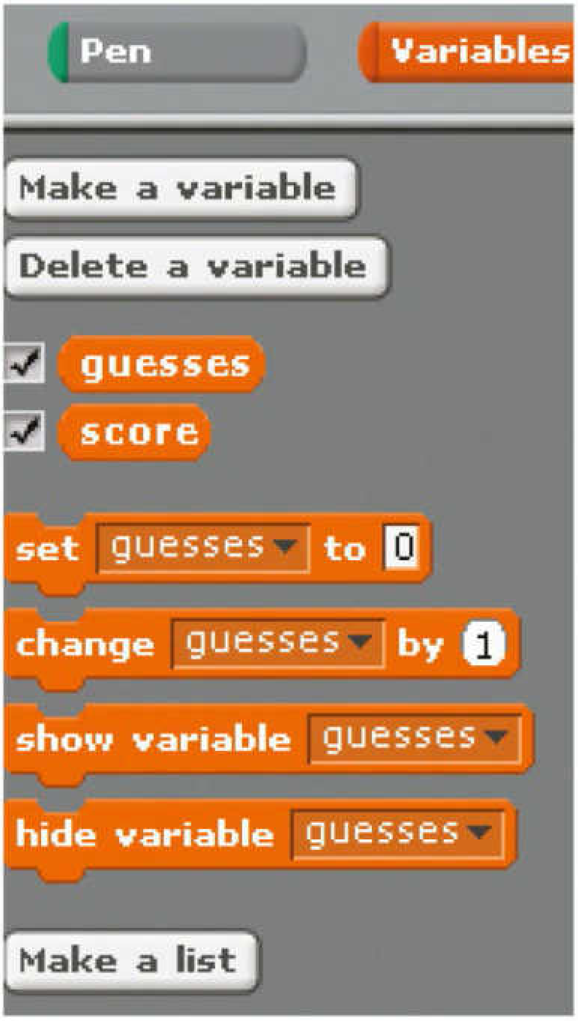


Fig 1.8 As you can see, there are two variables in our game: guesses and score.

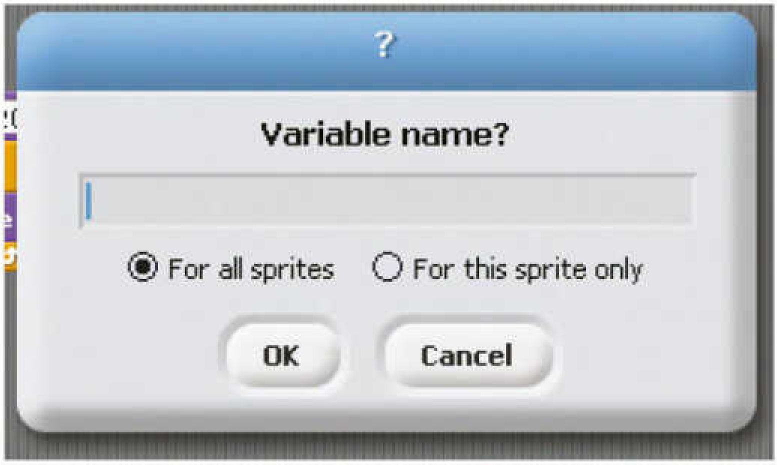


Fig 1.9 Hit the 'Make a Variable' button to create a variable.

```
a = 0
while a < 10:
    print a
    a = a + 1
```

So why did we include that piece of Python code in a Scratch tutorial? It simply illustrates that there isn't a lot of difference in the logic between the two languages, and that Scratch can be used to help understand the logic that powers many applications. In both Scratch and Python, we create a variable called **a** and set its value to be 0, from there we create a loop that will continue to iterate round until it reaches 9 (which is less than 10). Every time we go round the loop, we iterate **a** by 1, allowing us to count the number of times that we have been around the loop.

Testing our game

We've done it! We've made a game. So now let's play our game. The flow of the game should be as follows:

```
Click on green flag.
You will be asked for your name.
Matt says hello and Neil says hello and your name.
They both welcome you to the quiz.
Matt prompts Neil to ask the first question.
Neil asks a question.
A box will appear for you to type your answer.
If answer correct, then Neil will say so, Matt will also say something nice.
Your score will increase by one.
Else if your answer is wrong, you will be taunted by Matt and the number of guesses will increase by 1, leaving you with only 2 guesses left. You will then have another chance to answer the question.
If you guess correctly, you will move on to the next question, and this will happen twice as there are three questions.
If you answer all the questions correctly, Matt will tell you your final score and then say 'Game Over'.
The Game Over sprite will appear on screen and all of the scripts in the game will be turned off.
If the number of guesses made reaches 3 at any point in the game, then the game will automatically skip to the Game Over screen.
```

We're going leave Scratch for now, though this is just scratching the surface of what you can do with it. We're going to move back to using the more advanced Python on the Raspberry Pi over the next twenty or so pages. We hope you've enjoyed learning Scratch; it's a great tool to understand coding but it's also great fun to learn. ■

Coding in IDLE

Make Python coding more convenient and start hacking in Minecraft.

The rationale for favouring an integrated development environment (IDE) is discussed in the Mint section *(see page 28)*, wherein we met the *Geany* IDE. Ultimately, it comes down to being a more centralised, efficient way to work. Raspbian comes with IDLE – an IDE specifically designed for the Python language – and we're going to use it to further our programming adventures using a game called *Minecraft*, which you might have heard of.

You'll find *Minecraft* in the Games menu, so start it up now as a pre-coding warm-up. Click on Start Game and then Create New to generate a new world. After a few seconds, you'll find yourself in the *Minecraft* world. You can navigate with the W, A, S and D keys, and look around with the mouse. Space makes you (actually the *Minecraft* protagonist, whose name is Steve) jump, and double-tapping Space makes you fly (or fall, if you were already flying). You can bash away at the landscape with the left mouse button – the right button is used for placing blocks. To choose a type of block to place, press E to bring up a menu.

In the event that you're already a *Minecraft* aficionado, you've probably noticed that the Pi edition is slightly restricted. There's no crafting, no enemies and no Nether, but it does support networked play, and it is free. It also has another trick up its sleeve – it comes with a comprehensive API (application programming interface) for hooking it up to Python. So in no time you can be coding all kinds of colourful, improbable things involving obsidian and TNT blocks.

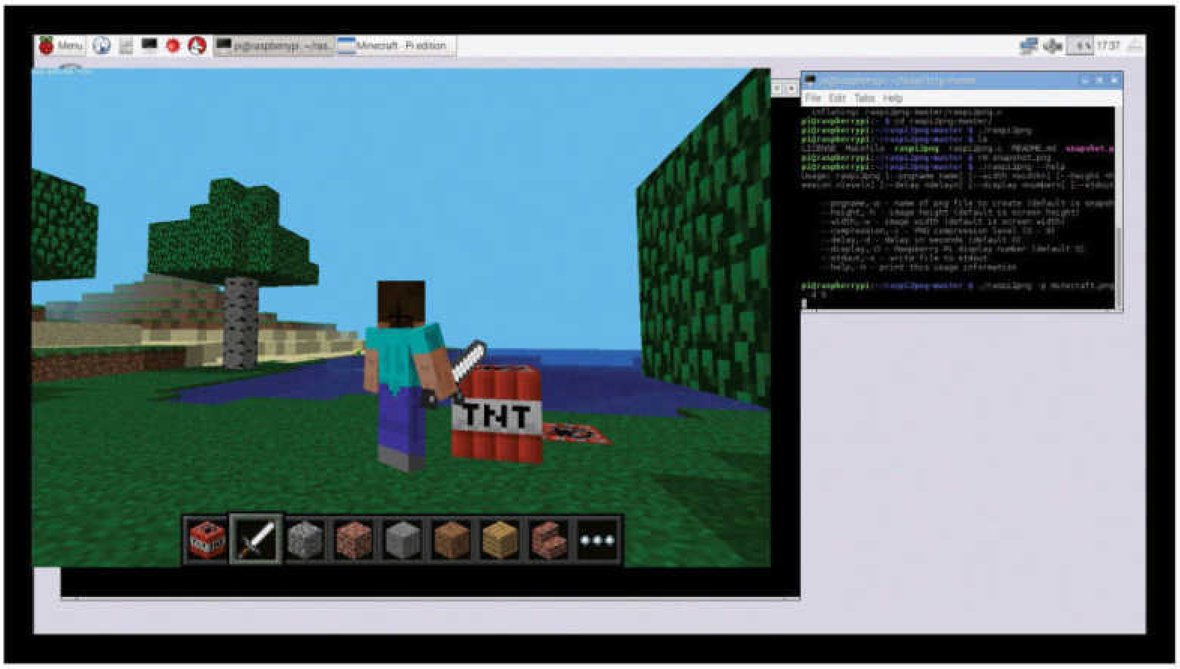
> **"In no time you can be coding all kinds of colourful, improbable things involving TNT blocks."**

IDLE hands

We'll put *Minecraft* aside for the moment as we introduce *IDLE*. Press Tab to release the mouse cursor from the game, and click on the Raspbian menu. You'll notice that it rather unhelpfully appears behind the *Minecraft* window. This is because of the selfish way the game accesses the GPU, in effect ignoring anything else on the screen and summarily drawing on top of it. So either minimise the *Minecraft* window or put it somewhere out of the way. We need it open so that *IDLE* can talk to it.

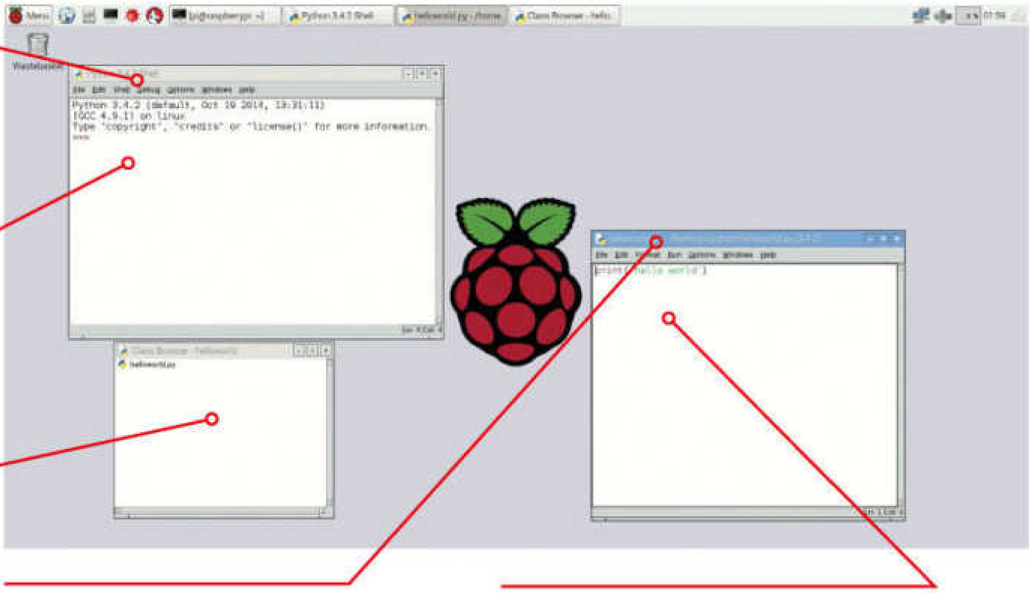
You'll find Python 2 and 3 versions of *IDLE* in the Programming menu. The *Minecraft* Python API has been updated to support version 3, so use that one. There's a quick guide to the interface opposite – note that only the Interpreter window opens initially.

Select File > Open, then navigate to the **python/** folder and open the **helloworld.py** file we made in the previous tutorial. The Editor window opens to display our code, complete with helpful syntax highlighting. From the Run menu, select Run Module or press F5. The Interpreter window now springs to life with its greeting. If you edit the code



Steve feels catastrophically compelled to introduce his sword to the TNT. It's actually harmless because it's not live. This can be changed, however...

The IDLE interface



```
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
```

```
print('hello world')
```

Debug

The built-in debugger helps you to track down any errors in your code. It also shows any variables, and navigates any breakpoints you've set up in the Editor.

Interpreter

We can use this shell exactly as we did earlier when we ran Python from the command line. Output from programs run from the Editor appear here.

Class browser

This is where the functions, classes or methods of any code we have open appear. You can access it from the Window menu.

Run menu

You'll find the Run Module option (F5) here. This checks the syntax of your code, then executes it, with the output appearing in the shell.

Editor

This is where we edit our code. Similar to *Geany (see page 28)*, Python keywords are highlighted. *IDLE* auto-indents using four spaces, rather than a tab.

without saving it, the Run Module helpfully asks if you want to save it first – you can't run unsaved code. If your code contains mistakes, you get an error message and the offending bit of code is highlighted in red.

Close the **helloworld.py** file and start a new one by choosing File > New from the Interpreter window. We'll start our new program with the boilerplate code required to get Python talking to our *Minecraft* game:

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
```

The first line imports the Minecraft module into our code – don't worry too much about the awkward syntax here, but do worry about getting the capitalisation correct. The second line sets up a new object, `mc` (short for minecraft), using the `create()` function that we imported in the first line. The `mc` object acts as a conduit between our code and the *Minecraft* game. Now let's add a third line, so that our program actually does something:

```
mc.postToChat('Hello Minecraft World')
```

Save this file in the **python/** folder as **hellomc.py**. Now arrange windows so that both your code and *Minecraft* are visible, and run your program. If all goes according to plan, a message should appear in the game. A few seconds later, it will vanish. We've manipulated the game's chat feature to display messages not from other users, but from our own program. And we can do an awful lot more.

As you [you mean Steve – *Ed.*] wander around the *Minecraft* world, you'll notice that some numbers in the top-left change. These give your position in terms of a 3D co-ordinate system. Don't worry if the idea of doing geometry is terrifying or offensive – for now, all you need to know is that the x and z directions are parallel to the world's surface, and the y direction determines your height.

The API contains functions for working with the player's position: we can use `mc.Player.getPos()` to get the player's current position, and `mc.player.setPos()` to change it. Add the following lines to your code:

```
x, y, z = mc.player.getPos()
mc.player.setPos(x, y + 50, z)
```

Save this update, and again make sure the *Minecraft* window is visible before running the module. As before, the message is displayed, but then we suddenly find ourselves high in the sky and falling fast. Fortunately, Steve doesn't get hurt in the game, and he'll land somewhere around where he was before the program was run, presumably a little discombobulated, though…

Catching the bug

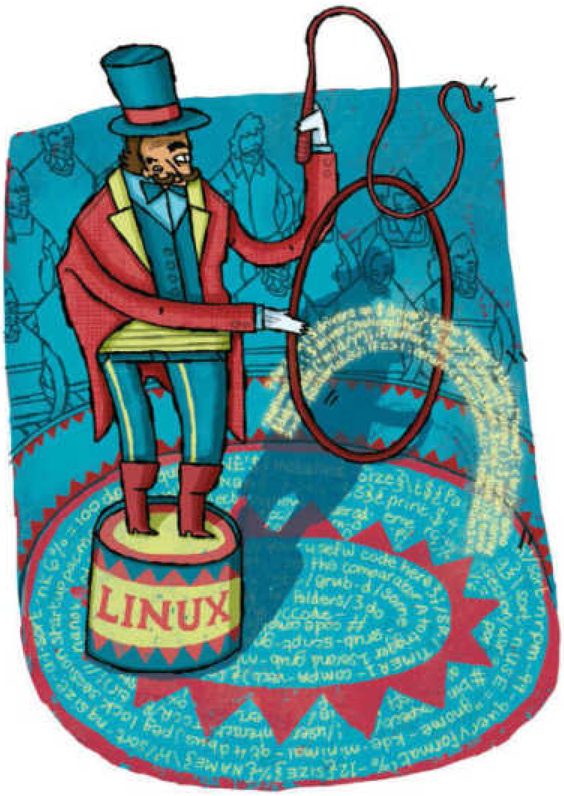
We can use *IDLE's* built-in debugger to see this happening step by step. Right-click the final `setPos()` line and choose Set Breakpoint. Next, in the Interpreter window, choose Debug > Debugger. Now return to your program and run it once again. The debugger enables you to run step by step through your code, which is incredibly useful for (funnily enough) ridding your code of bugs.

By defining a breakpoint, we can also tell *IDLE* to execute everything up to this line, and on our go ahead continue. From the debugger window, choose Go. Notice that in the game window we get as far as displaying the message, but not as far as being sent skyward. There's a lot of stuff you needn't worry about going on in the debugger, but notice that the variables `x`, `y` and `z` from the `getPos()` line have been calculated in the Globals panel. Press Go again to initiate Steve's catapulting skyward.

The *Minecraft* API also contains `getBlock()` and `setBlock()` functions for probing and manipulating the environment. These functions are beyond the scope of this primer, unfortunately, but you'll be able to find some great *Minecraft* tutorials at Martin O'Hanlon's handy website: **http://stuffaboutcode.com**. ■

Python 3: Go!

Learning to program with Python needn't involve wall-climbing or hair-tearing – just follow our three guides to Python prowess.



After the basics let's delve into Python. Learning a new programming language can be as daunting as it is rewarding, all the more so for those embarking on their maiden coding voyage. Exactly which language is most suitable for beginners can be debated until the bovine herd returns, but Python is certainly a worthy candidate. Python code is easy to follow, the syntax favours simplicity throughout and its enforcement of indentation encourages good coding styles and practices. Your distribution probably already has some version of Python installed, but for this tutorial we're targeting the newer version 3. You can check your Python version by opening up a terminal and typing:

```
$ python -V
```

Many distributions ship with both versions installed, but some (including Debian 8) still default to the 2.7 series. If the previous command indicated that this was the case, see what happens if you do:

```
$ python3
```

If that doesn't work, then you'll need to find the **python3** packages from your package manager. They should be straightforward to locate and install. If, on the other hand, the command succeeded, then you will, besides being availed of version information, find yourself at the Python 3 interpreter. Here you can execute code snippets on the fly, and here is where we will begin this tutorial. You can exit the interpreter at any time by pressing Ctrl+D or by typing:

```
>>> quit()
```

However, your first lines of code really ought to be more positive than that, so let's instead do a cheery:

```
>>> print('Hello world.‘)
```

Having pressed Enter and greeted our surroundings, we can get down to some proper coding. It's useful to accept and work with user input, which is accomplished as follows:

```
>>> name = input(‘State your name ’)
```

Note the space before the closing quote. The command prints the given prompt and waits for something to be typed, which will appear alongside. This means that when the user starts typing their name (or whatever else they want), it is separated from our curt demand. We could also have used a **print()** line to display this prompt, and then instead used a blank input call as follows:

```
>>> name = input()
```

This means that input is accepted on a new line. Either way, the user's input is stored as a string in a variable called **name**. From the interpreter, we can see the values of any variable just by typing its name – however, we can also use the **print** function:

```
>>> print(‘Greetings’, name, ‘enjoy your stay.’)
```

The **print()** function can work with as many arguments as you throw at it, each one separated by a comma. Note that we don't need to put spaces around our variable, as we did with the **input** function; by default, **print** separates arguments with a space. You can override the behaviour by specifying a **sep** parameter to the function – for example, the following code uses a dot:

```
>>> print(‘Greetings’, name, ‘enjoy your stay.’,sep=‘.’)
```

On the other hand, using **sep=‘’** instead gives no separation at all. Besides separators, **print()** also issues a newline character, represented by the string **‘\n’** after the final argument. This can be easily changed, though, by specifying the **end** parameter, which is sometimes desirable, depending on the circumstances.

Besides welcoming people, we can use the interpreter as a calculator. It understands **+** (addition), **-** (subtraction), ***** (multiplication), **/** (division) and ****** (exponentiation), as well as many more advanced maths functions if you use the maths module. For example, to find the square root of 999 times 2015 and store the result in a variable **x**:

```
>>> import math
>>> x = math.sqrt(999 * 2015)
```

One of many things that Python helpfully takes care of for us is data types. Our first variable **name** was a string, while the recently defined **x** is a floating point number (or **float** for short). Unlike typed languages, where we would have to explicitly specify the type of a variable where it is defined, Python is smart enough to figure out that sort of information for itself, saving us the trouble. Also, Python variables do not have fixed types, so we could actually recast **x** above to an integer with:

```
>>> x = int(x)
```

Raspberry Pi-thon

If you're following this tutorial on a Raspberry Pi (1 or 2) and using the Raspbian distribution, there's good news: you don't need to perform any additional steps to get Python working. In fact, you get the pleasure of a ready-to-roll development environment called *IDLE*. You might find this more fun to work with than the command-line interpreter we use throughout this tutorial, and such things certainly become advantageous when working with larger projects.

IDLE enables you to write and edit multiple lines of code as well as providing a REPL (Read Evaluate Print Loop) for immediate feedback. To see this in action, select New Window, hammer out a few lines of code and then run them by pressing F5 or selecting Run Module from the Run menu.

Newer versions of Raspbian come with both versions 2 and 3 of Python, together with two separate *IDLE* shortcuts for starting each version. Previous Raspbian versions came with only the former, but if you've done an **apt-get dist-upgrade** recently, you'll probably find that the new version has been pulled in. But if you don't have Python 3 installed (in other words running **python3** returns **command not found**, then you can get it by opening *LXTerminal* and issuing the following commands:

```
$ sudo apt-get update
$ sudo apt-get install python3
```

Now **x** has been rounded down to the nearest whole number. We could convert it to a string as well, using the **str** type. Another important construct in Python is the list. A list is defined by using square brackets and may contain any and all manner of data – even including other lists. Here's an arbitrary example:

```
>>> firstList = ['aleph', 'beth', 3.14159265]
```

Items in our list are zero-indexed, so we access the first item, for example, with **firstList[0]** and the third one with **firstList[2]**. Some languages are one-indexed, which some people find more intuitive, but Python is not one of them. Strings are similar to lists (in that they are a sequence of characters) and many list methods can be applied to them. Thus we can get the first character of **name** with **name[0]**. We can also get the last character using **name[-1]** rather than having to go through the rigmarole of finding the string's length first, for which you would have to use the **len()** function. Strings are immutable – which means that once they've been defined, they cannot be changed – but this is fine because they can be redefined, copied or reconstructed depending on our purposes.

A particularly useful construct is **list** (or string) slicing. We can get the first two elements of our list by using **firstList[:2]**, or all but the first item with **firstList[1:]**. These are short forms of the more general slicing syntax **[x:y]**, which returns the substring starting at position **x** and ending at position **y**. Omitting **x** defaults to the beginning of the list, and omitting **y** defaults to the end. One can even specify a third parameter, which defines the so-called step of the slice. For example, the slice **[x:y:2]** gives you every second item starting at position **x** and ending at position **y**, or wherever the previous step lands. Again, this can be abbreviated – if you just want every second item from the whole list – to **[::2]**. Negative step sizes are also permitted, in which case our starting point **x** is greater than **y**, and omitting **x** defaults to the end of the list. Thus slicing our list above like so returns the reverse of that list:

```
>>> firstList[::-1]
```

Besides defining lists by explicitly specifying their members, there are a number of constructs available for defining lists that have some sort of pattern to them. For example, we can make the list **[0,1,2,3,4,5]** by using **list(range(6))**. In Python 3, the **range()** function returns an iterator (it doesn't contain any list items but knows how to generate them when given an index), so there is an additional function call to turn it into a list proper. Again, there is potential for 0-based grievances, because 6 is not in that list, and again it's something we simply have to get used to. Similar to slicing, the **range()** function can take optional arguments specifying the start value and step size of the range. This means that we can get all the odd numbers between 5 and 19 (including the former but excluding the latter) by using:

```
>>> twostep = list(range(5,19,2))
```

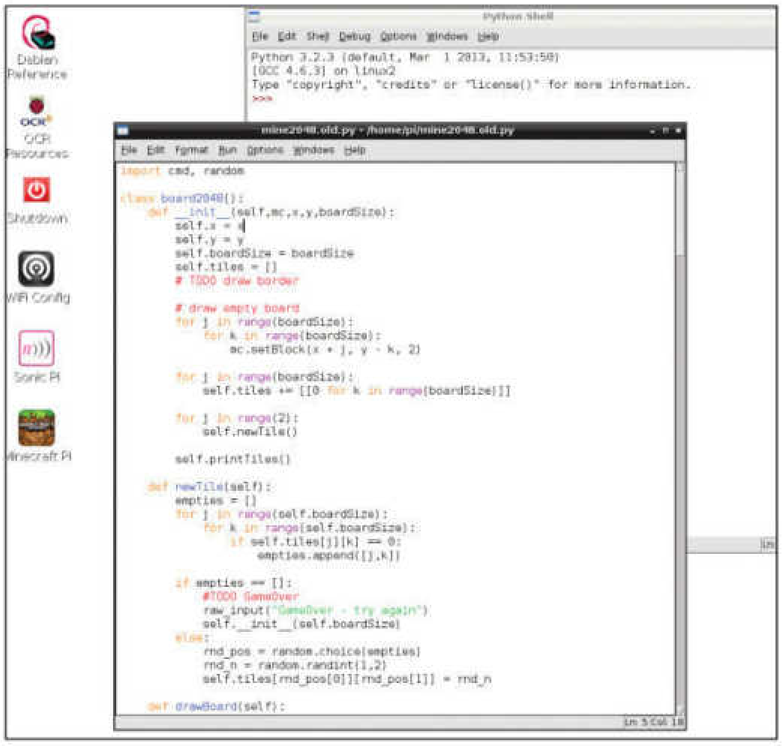
Negative step sizes are also supported, so we could count down from 10 to 1 with **range(10,0,-1)**.

Items can be inserted into lists – or removed from them – by using various methods. Methods in Python are properties of an object, and are called by suffixing said object with a dot, followed by the method name. So we could add **19** to the end of our previous list by using the **append()** method, such as:

```
>>> twostep.append(19)
```

We could also insert the value **3** at the beginning of the list with **twostep.insert(0,3)**. There are many other list methods, which you can peruse using **help(list)**. Help is available for any Python keyword and many other topics, and this can be a very useful resource when you get stuck.

When you start working with multiple lists, you might witness some rather strange behaviour, such as the diagram »


Debian Reference
OCR Resources
Shutdown
WiFi Config
Sonic Pi
Minecraft Pi
Python Shell
File Edit Shell Debug Options Windows Help
mine2048.old.py - /home/pi/mine2048.old.py
File Edit Format Run Options Windows Help
Ln: 5 Col: 18

```
Python 3.2.3 (default, Mar  1 2013, 11:53:50)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
>>>
```

```
import cmd, random

class board2048():
    def __init__(self,mc,x,y,boardSize):
        self.x = x
        self.y = y
        self.boardSize = boardSize
        self.tiles = []
        # TODO draw border

        # draw empty board
        for j in range(boardSize):
            for k in range(boardSize):
                mc.setBlock(x + j, y - k, 2)

        for j in range(boardSize):
            self.tiles += [[0 for k in range(boardSize)]]

        for j in range(2):
            self.newTile()

        self.printTiles()

    def newTile(self):
        empties = []
        for j in range(self.boardSize):
            for k in range(self.boardSize):
                if self.tiles[j][k] == 0:
                    empties.append([j,k])

        if empties == []:
            #TODO GameOver
            raw_input("GameOver - try again")
            self.__init__(self.boardSize)
        else:
            rnd_pos = random.choice(empties)
            rnd_n = random.randint(1,2)
            self.tiles[rnd_pos[0]][rnd_pos[1]] = rnd_n

    def drawBoard(self):
```

› **The *IDLE* environment is ideal *[ha, ha – Ed]* for developing larger Python projects. It's included in the Raspbian distribution, too.**

» opposite illustrates. This seemingly inconsistent behaviour is all to do with how variables reference objects internally – variables are really just labels, so many variables can point to the same thing.

Very often, the canny coder makes extensive use of loops, wherein a codeblock is iterated over until some condition is met. For our first loop, we're using the **for** construct to do some counting. Note the indentation here; things inside a **for** loop (or indeed any other block definition) must be indented, otherwise you get an error. When you start such a block in the interpreter, the prompt changes from **>>>** to **...**. Python permits any number of spaces to be used for indentation, but consistency is key. We (alongside many others) like to use four spaces.

```
>>> for count in range(5):
...         print('iteration #', count)
```

Enter a blank line after the **print** statement to see the stunning results of each iteration in real time.

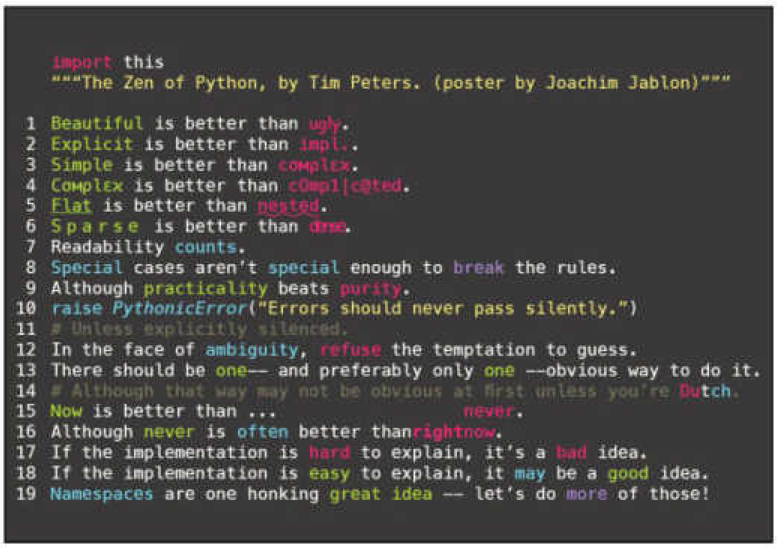
The variable **count** takes on the values **0** to **4** from the **range** iterator, which we met earlier. The result is that we issue five **print** statements in only two lines of code. If you want to iterate over different values for **count**, then you can use a different **range** or even a list – you can loop over any objects you like, not just integers.

Another type of loop is the **while** loop. Rather than iterating over a range (or a list), our wily **while** loop keeps going over its code block until some condition ceases to hold. We can easily implement **for** loop functionality with this construct, as shown here:

```
>>> count = 0
>>> while count < 5:
...       print(count)
...       count += 1
```

The **print** statement belongs to the loop, thanks to the indentation. The code still runs if you don't indent this last line, but in that case the **print** call is not part of the loop, so only gets executed at the end, by which time the value of **count** has reached **5**. Other languages delineate code blocks using brackets or braces, but in Python it's all colons and judicious use of white space. That **while** loop was rather trivial, so let's look at a more involved example:

```
>>> year = 0
>>> while year < 1900 or year >= 2015:
...       year = input("Enter your year of birth: " )
...       year = int(year)
```



```
import this
"""The Zen of Python, by Tim Peters. (poster by Joachim Jablon)"""

Beautiful is better than ugly.
Explicit is better than impl.
Simple is better than complex.
Complex is better than c0mpl|c@ted.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
raise PythonicError("Errors should never pass silently.")
# Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
# Although that way may not be obvious at first unless you're Dutch.
Now is better than ...                never.
Although never is often better than rightnow.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

› **This Pythonic creed is worth studying. There are all kinds of bad programming habits, which are best to avoid from day zero.**

We met the input statement on the first page, so this code just keeps asking you the same thing until an appropriate value is selected. We use the less than (**<**) and greater than or equal to (**>=**) operators conjoined with an **or** statement to test the input. So long as **year** has an unsuitable value, we keep asking. It is initialised to **0**, which is certainly less than **1900**, so we are guaranteed to enter the loop. You could change **1900** if you feel anyone older than 115 might use your program. Likewise, change **2015** if you want to keep out (honest) youngsters.

Whenever you deal with user input, you must accept the possibility that they will enter something not of the required form. They could, for example, enter their granny's name, or the first line of *The Wasteland,* neither of which can be interpreted as a year. Fortunately, the last line of our example does a good job of sanitising the input – if we try to coerce a string consisting of anything other than digits to an int, then we end up with the value **0**, which guarantees that we continue looping the loop. Note that the **input** function always returns a string, so even good input – for example, **'1979'** – needs some treatment; trying to compare a string and an int results in an error.

Grown-up coding

It's time for us to level up and move on from the interpreter – even though it's awfully handy for checking code snippets, it is not suitable for working on bigger projects. Instead we'll save our code to a text file with a **.py** extension, which we can either import into the interpreter or run from the command line. You need to find a text editor on your system – you can always use *nano* from the command line, but you may prefer to use a graphical one, such as Gnome's *gedit,* KDE's *kate* or the lightweight *Leafpad,* which comes with Raspbian. You may even wish to use a development environment such as *IDLE* (which comes with Raspbian and is designed especially for the language) but we don't require any of its many features for this introduction, so just a simple text editor will suffice.

Having found such a thing, enter the following listing:

```
#! /usr/bin/env python3
# My first Python 3 code.
# The # symbol denotes comments so anything you put here
doesn't matter

import datetime

def daysOld(birthday):
    today = datetime.date.today()
    ageDays = (today - birthday).days
    return(ageDays)


if __name__ == '__main__':
    weekdays = ['Monday', 'Tuesday', 'Wednesday',
'Thursday', 'Friday', 'Saturday', 'Sunday']

    birthyear = int(input('Enter your year of birth: '))
    birthmonth = int(input('Enter your month of birth: '))
    birthdate = int(input('Enter your date of birth: '))

    bday = datetime.date(birthyear, birthmonth, birthdate)
    dayBorn = weekdays[bday.weekday()]

    print('Hello, you were born on a', dayBorn)
```

```
 print(‘and are’, daysOld(bday),‘days old.’)
```

Save the file as **~/birthday.py** (Python programs all have the **.py** extension). Before we analyse the program, you can dive straight in and see it in action. Almost, anyway – first we must make it executable by running the following command from the terminal, then we can call it into action:

```
$ chmod +x ~/birthday.py
$ ~/birthday.py
```

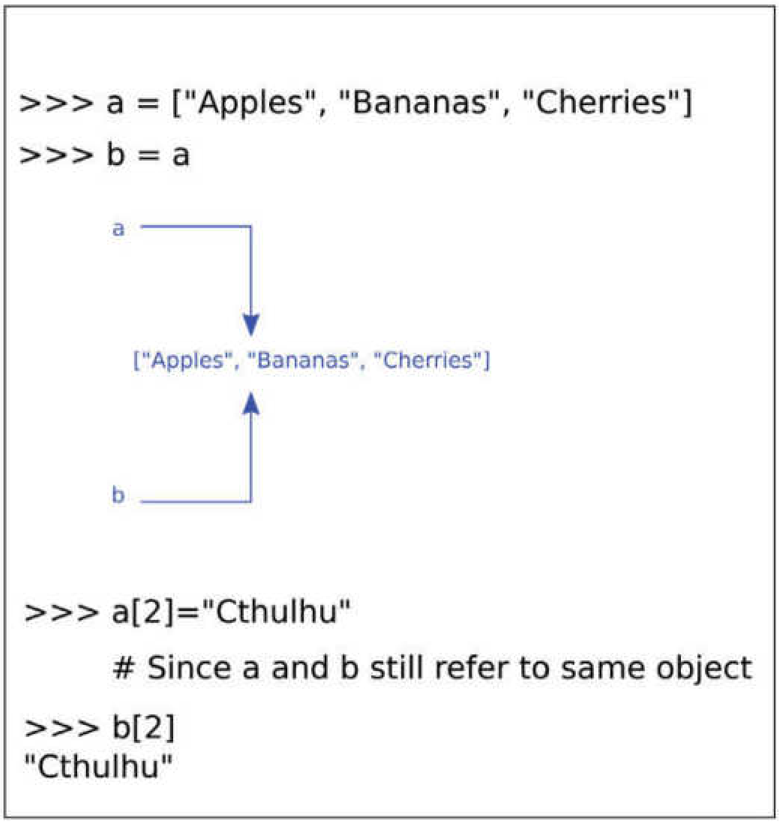
If you entered everything correctly, then the program works out how many days old you are and on which weekday you were born. Due to leap years, this would be some pretty fiddly calendrics if we had to work it out manually (though John Conway's Doomsday algorithm provides a neat trick). As well as having a simple core language structure, Python also has a huge number of modules (code libraries), which also strive for simplicity. This example uses the datetime module, which provides all manner of functions for working with dates and times, oddly enough.

We'll do a quick rundown of the code, because there are a couple of things that we haven't seen before. The first line just tells *Bash* (or whatever shell you use) to execute our script with Python 3. Next, after the comments, we import the datetime module and then begin a new codeblock. The **def** keyword is used to define a function – we've used plenty of functions in this tutorial, but now we're making our own. Our function takes one parameter, **birthday**, and uses the datetime module to do all the hard work, returning the number of days the user has been alive.

The strange **if** statement is one of Python's uglier constructions. The special variable **__name__** takes on the special value **__main__** when the code is run from the command line, as opposed to being **import**-ed into another program or the interpreter. We have written a bona fide Python module here, but when we import it, everything in the **if** block is ignored.

The next four lines are similar to what we've come across before. To keep things simple, we haven't bothered with list comprehension for the weekdays, and we trust the user to provide sensible dates. You are free to remedy either of these using your newfound skills.

The datetime module provides a special data type – **datetime.date** – and our next line (using the input that we have just harvested) sets our variable **bday** to one of this species. Date objects have various methods, including **weekday()**, which we use next. This returns an integer between **0** and **6**, with **0** corresponding to Monday and **6** to Sunday, so using this as an index into our list **weekdays** in the next line gives the required string. The final line calls our function from inside the **print()** function. Besides methods such as **weekday()** and **year()**, **date** objects also allow for general date arithmetic to be done on them. Therefore we can subtract two dates, which returns a **datetime.timedelta** object, which we can then convert to an integer using **days()** or **seconds()**.



```
>>> a = ["Apples", "Bananas", "Cherries"]
>>> b = a
```

```
>>> a[2]="Cthulhu"
        # Since a and b still refer to same object
>>> b[2]
"Cthulhu"
```

› You might run into this apparently spooky action at a distance when working with lists. It's not the work of Chthonic demons, however, it's just how Python rolls.

And so concludes the first of our Python coding tutorials on the Pi. We've actually covered a good deal of the core language constructs, and with a little imagination it's not difficult to develop the ideas we've introduced into a larger and more exciting project. Help is never far away, though, and you can find some great Python tutorials on the web. The official tutorial (**https://docs.python.org/3/tutorial/**) is quite extensive, and you can find others at **http://tutorialspoint.com** and **www.codecademy.com**.

We also have plemnty more excellent tutorials in this very publication. However, learning to code is much more about experimentation and dabbling than following instructions. No tutorial can teach this, but hopefully we've sufficiently piqued your curiosity that you will continue your coding adventures. ■

List comprehensions

Lists and the various constructs we've just introduced can join forces to form one of Python's most powerful features: list comprehensions. These are a great demonstration of Python's laconic concision. Consider the following example:

```
>>> daysShort = [‘Mon’, ‘Tues’, ‘Wednes’,
‘Thurs’, ‘Fri’, ‘Satur’, ‘Sun’]
>>> days = [j + ‘day’ for j in daysShort]
```

Very often, coders use short variable names, commonly **i**, **j** and **k**, for ephemeral variables, such as those which are used in loops or comprehensions. Our iterator variable **j** runs over the beginnings of the names of each day of the week, and then our list comprehension suffixes the string **day** – the addition operator (**+**) concatenates (in other words, joins) strings together. If you want a more arithmetical example, you could try:

```
>>> [j ** 3 for j in range(11)]
```

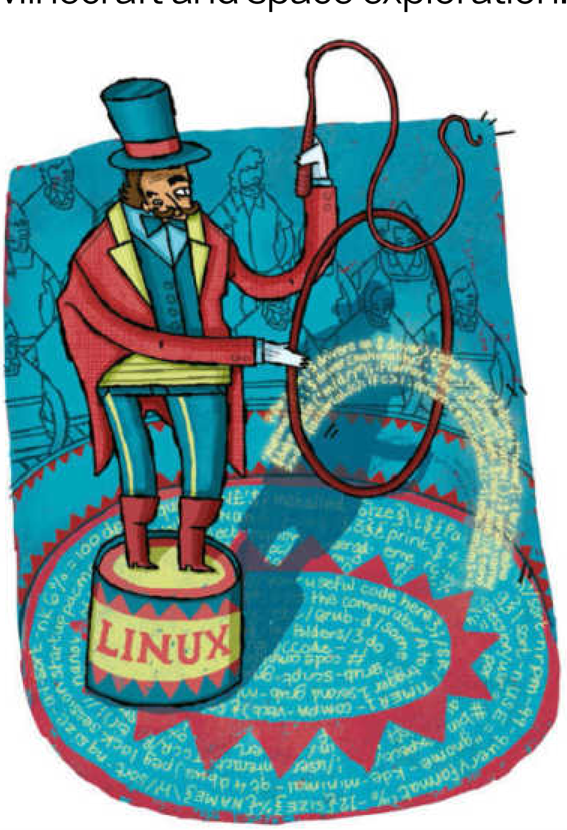
This returns a list comprising the cubes of the numbers from 0 to 10. Comprehensions can use all kinds of other constructs as well as **for** loops. For example, if we have a list of names, we could select only those beginning with ‘Z’ or those which have ‘e’ as their second letter by using:

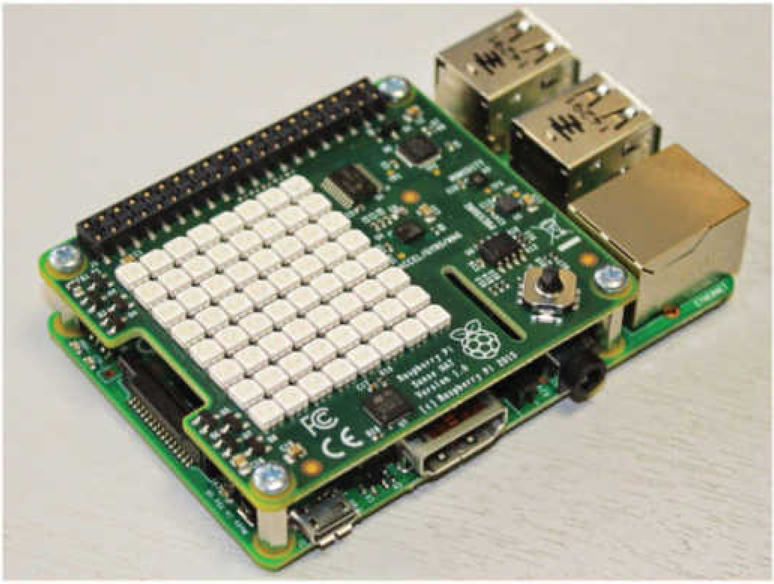
```
>>> names = [“Dave”, “Herman”, “Xavier”,
“Zanthor”]
>>> [j for j in names if j[0] == ‘Z’ or j[1] == ‘e’]
[‘Herman’, ‘Zanthor’]
```

It's worth noting at this point that to test for equality, we use the **==** operator. Be careful, because a single equals sign is only used for assignment, so doing something such as **if a = 5:** would result in an error.

Astro Pi and Minecraft

We conclude our Python exposition on a recreational note, combining Minecraft and space exploration.





❯ **Astro Pi sits snugly atop Terrestrial Pi, connecting via the GPIO pins.**

We have published a fair few guides to the Python API for *Minecraft:Pi* edition for the Raspberry Pi, see page 108. They're not critical for understanding this tutorial, but are certainly worth checking out if the idea of programming a voxel-based world appeals. Also worthy of attention, and also featuring excellent *Minecraft* tutorials, is Martin O'Hanlon's excellent website **http://stuffaboutcode.com**, whence comes the code used in this tutorial. The API, besides being a lot of fun, further edifies the Pi's status as an educational tool, because many young learners find programming much more intuitive when the results can be visualised three-dimensionally from the point of view of Steve, the game's intrepid hero.

One of the most exciting Raspberry Pi projects right now is the Astro Pi. This is a Hardware Attached on Top (HAT) expansion board, which features all manner of cool instrumentation: gyroscope, accelerometer, magnetometer, various environmental sensors, a multicoloured LED matrix and much more. It is available for free for school pupils and educators from the website **http://astro-pi.org**, if you can come up with a good coding idea, and will very soon be available to the general public. The most exciting thing about the project is that in November 2015, a Raspberry Pi, attired with an Astro Pi, will join ESA astronaut Tim Peake for a six-month stay on board the ISS. A competition is currently underway across UK schools in which entrants must devise Python code utilising the Astro Pi. The lucky winners will have their programs join Tim on his voyage to the ISS, where they will be run in zero-G. Owing to power requirements aboard the space station, the Pi cannot be connected to a proper display – its only means of visual feedback is the 64-LED array. This is multicoloured, though, so can still provide a fair amount of information.

The Astro Pi ships with a Raspbian SD card preloaded with all the required drivers and configs, but you can also download an installer for these. Open a terminal and type:

```
$ wget -O - http://www.raspberrypi.org/files/astro-pi/astro-pi-install.sh --no-check-certificate | bash
```

It takes a while to run on the original, single-core Pi, and you'll need to reboot for everything to work properly. While we're setting things up, make sure you've got *Minecraft* and the Python 3 API set up too. These are installed by default on recent Raspbian versions (there should be an icon on your desktop).If you don't have it, install it from the repositories:

```
$ sudo apt-get update
$ sudo apt-get install minecraft-pi
```

If you don't have an Astro Pi board, you can still have a lot of fun playing with the *Minecraft* Python API. Just start *Minecraft,* enter a new world, then Alt+Tab back to the

The mission

On Thursday 19 November 2015, two Raspberry Pis (one with the infrared filtered Pi-Noir camera module, and one with the normal Pi-Cam) will join astronaut Major Tim Peake aboard a Soyuz rocket, blasting off from Baikonur cosmodrome in (glorious nation of) Kazakhstan.

A nationwide coding competition was launched for primary and secondary school pupils, with the winners having Tim run their code when he arrives at the International Space Station. Space, industry and education sectors came together to make this exciting outreach project possible. A number of themes have been identified to inspire junior coders: Space Measurements, Spacecraft Sensors, Satellite Imaging and Remote Sensing, Space Radiation and Data Fusion.

Getting the Raspberry Pi and Astro Pi approved for cargo is a complex process – anything that is going to be connected to the ISS power supply has to be thoroughly tested. There are all manner of considerations, including electromagnetic interference, sharp edges assessment, and thermal testing – in zero-G conditions, air warmed by the Pi's CPU won't dissipate (as happens naturally by convection here on Earth), instead it will just hover around being hazardous. The flight case has been precision-engineered to comply with rigorous vibration and impact tests, as well as to provide much needed airflow. It also acts as a giant heatsink for the Pi's CPU – surfaces on the ISS are not allowed to exceed 45°C.

desktop and open a terminal window. From here, start Python 3 (**$ python3**) and enter the following:

```
>>> from mcpi import minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.postToChat("Hello world.“)
```

The API is capable of much more than this – we can change the player and camera position, get the ground level at a given set of co-ordinates, and get and set block information there, too. The following draws a rough and ready representation of a Raspberry Pi (it could be a Pi 2 or a Model B+), component by component:

```
>>> mc.setBlocks(-6, -3, -9, 7, -3, 11, 35, 13)
>>> mc.setBlocks(7, -2, -8, 7, -1, 5, 35, 15)
>>> mc.setBlocks(4, -2, 8, 6, 0, 11, 42)
>>> mc.setBlocks(0, -2, 8, 2, 0, 11, 42 )
>>> mc.setBlocks(-5, -2, 8, -2, 0, 11, 42)
>>> mc.setBlocks(-5, -2, 1, -2, -2, 1, 35, 15)
>>> mc.setBlocks(2, -2, -9, -1, -2, -9, 35, 15)
>>> mc.setBlocks(-6, -2, -7, -6, -2, -6, 42)
>>> mc.setBlocks(-6, -2, -2, -6, -2, 0, 42)
>>> mc.setBlock(-6, -2, 3, 35, 15)
>>> mcsetBlocks(0, -2, -2, 2, -2, -4, 35, 15)
```

The **setBlocks()** function fills the cuboid given by the first six numbers (the first three are co-ordinates of one corner and the next three are co-ordinates of the other corner). The next number decides the block type (42 stands for iron, while 35 stands for wool). Wool is a special block that comes in 16 colours; we don't need to specify a colour, but if we wish to do so, we can through an optional eighth parameter, which in general is referred to as block data. We've used the **setBlock()** function above – this just sets up a single block, so only requires one set of co-ordinates, plus the block type and optional block data arguments. You can find a thorough guide to the API at **www.stuffaboutcode.com/p/minecraft-api-reference.html**, but we'll describe what everything does as we use it.

For now, we'll turn our attention to coding on the Astro Pi. This connects to the Pi via the General Purpose Input/Output (GPIO) pins, but thanks to the astro_pi Python module, we don't need to worry about coding at the level of individual pins. The module provides some simple and powerful functions for querying sensors and manipulating the LED array. For example, we can display a suitably space-themed message in an extraterrestrial shade of green using the **show_message** function:

```
from astro_pi import AstroPi
ap = AstroPi()
ap.show_message(“E.T. phone home…”, text_colour = [0
,255,0])
```

The **text_colour** parameter here specifies the RGB components of the colour. As well as text, we can also work with the individual LEDs, using the **set_pixels()** function. This probably isn't the type of thing you want to do too much of by hand, though, because each LED requires its own colour triple. You can use this trick as a slight shortcut if you're only going to be working with a few colours, though:

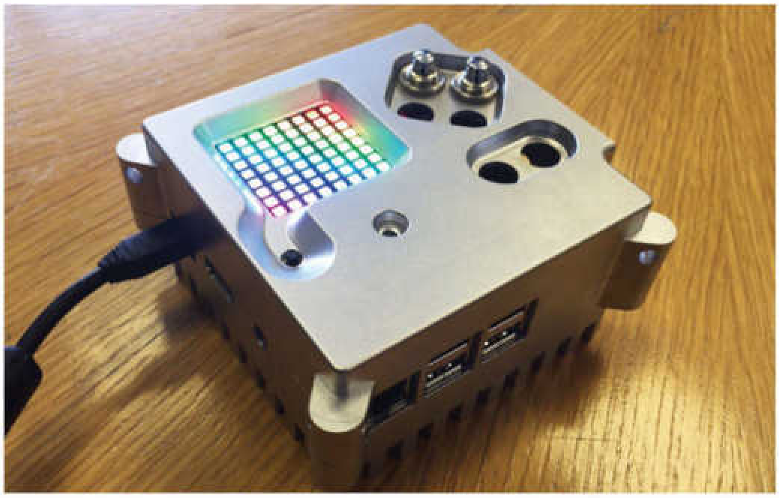
```
X = [255, 0, 0]  # Red
O = [255, 255, 255]  # White

question_mark = [
O, O, O, X, X, O, O, O,
O, O, X, O, O, X, O, O,
O, O, O, O, O, X, O, O,
O, O, O, O, X, O, O, O,
O, O, O, X, O, O, O, O,
O, O, O, X, O, O, O, O,
O, O, O, O, O, O, O, O,
O, O, O, X, O, O, O, O
]

ap.set_pixels(question_mark)
```

It is probably more desirable to type this code into a text editor and then import it into Python, rather than work in the interpreter. So you need to put the first two lines from our E.T. phone home snippet at the top of the program, so that our **ap** object is correctly set up. You can then import it into the interpreter or run it from the command line if you prefer. As well as dealing with individual pixels, we can also load images

»

Quick tip

We're assuming you use Python 3 throughout this series. This project still works with Python 2, but you may need to install the Pillow imaging library with **sudo pip install Pillow**.



Space case. This carefully machined and very strong case (it's made of 6063 grade aluminium, don't ya know) will house the Pi on its space odyssey.

» directly on to the array by using the **load_image()** function as follows:

```
ap.load_image("~/astro-pi-hat/examples/space_invader.png")
```

Bear in mind the low resolution here – images with lots of detail don't work very well, but it's excellent for displaying pixel art and the like. Also, the LEDs are only capable of displaying 15-bit colour depth (five bits for red, six for blue, and five for green), so colours are dithered accordingly.

We can also query the Astro Pi's many inputs using the self-explanatory functions **get_humidity()**, **get_temperature()** and **get_pressure()**. The gyroscope, accelerometer and magnetometer are all integrated into a so-called Inertial Measurement Unit (IMU). We won't be doing any inertial measuring, but you can find all the relevant API calls in the documentation at **https://github.com/astro-pi/astro-pi-hat/blob/master/docs/index.md**. You'll also find some great examples in the **~/astro-pi/examples** directory, which show off everything from joystick input to displaying a rainbow pattern. Run these with, for example:

```
$ cd ~/astro-pi-hat
$ sudo python3 rainbow.py
```

The virtual interactive Astro Pi

The project we're going to undertake is actually the brainchild of student Hannah Belshaw, who submitted the idea to the Astro Pi competition. The (extensive) coding was done by Martin O'Hanlon, to whom we're ever so grateful. The project draws a Raspberry Pi (like we did earlier), but this one is equipped with the Astro Pi HAT.

Rather than have you copy out the code line by line, we'll download it straight from GitHub, and explain select parts of it. Don't expect to understand everything in the code right away – there's a lot of stuff we haven't covered – but once you get a couple of footholds, it'll serve as an excellent base for adding your own ideas. To download the code to your home directory, just open up a terminal and issue:

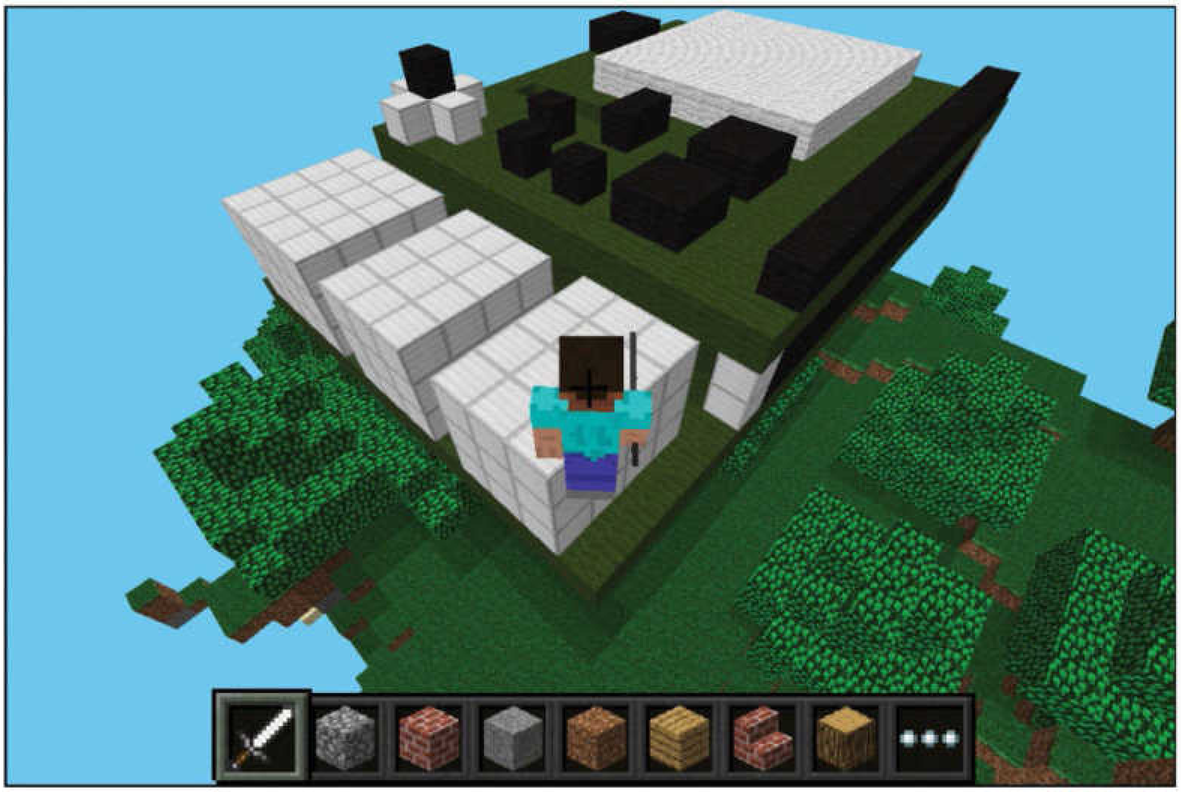
```
$ cd ~
$ git clone https://github.com/martinohanlon/
MinecraftInteractiveAstroPi.git
```

This downloads all the required project files into the **~/MinecraftInteractiveAstroPi/** directory. Before we run it, make sure *Minecraft* is started. Also, if you've still got a Python session open from the beginning of the tutorial, then exit it now (Ctrl+D) so that things don't get confused. Now we're good to go. Because the Astro Pi libraries need GPIO access, and this in turn requires root privileges, we need to run it through **sudo**:

```
$ cd ~/MinecraftInteractiveAstroPi
$ sudo python mcinteractiveastropi.py
```

Everything in *Minecraft* is located by co-ordinates – triples of numbers which identify tiles by their x, y and z positions. The y co-ordinate measures height, x and z are harder to define, but they are nonetheless at right angles to each other and go in directions parallel to the ground plane. Normally, the player will start at position 0,0,0 (you can see your co-ordinates in the upper-left corner), in which case you might notice that it suddenly got dark in *Minecraft*-world. If you look up, you will see that this partial eclipse is caused by the appearance of a blocky contraption that our code has created. *Minecraft* allows you to fly by tapping and then holding Space. So fly up to this and you'll see an introduction message telling you to hit the virtual Astro Pi by right-clicking it with your sword. Left-clicking results in Steve hitting stuff properly, which would destroy the blocks that constitute our virtual device, so don't do that.

You can explore all the components of the Pi and the Astro Pi. Hitting the sensors, the latter avails you of the local pressure, temperature and humidity. You can also find out the orientation of the Astro Pi, which is given in terms of the pitch, roll and yaw angles. Bear in mind that you need to have calibrated the magnetometer for these readings to make sense. Instructions for this are at **www.raspberrypi.org/forums/viewtopic.php?t=109064**. You can also use the virtual joystick to move the virtual assemblage in three



› **Sword-wielding Steve stands ominously close to a USB port. Something about knives and toasters...**

dimensions: the button toggles whether it moves in the horizontal or vertical plane.

Studying the code

If you open up the main project file, **mcinteractiveastropi.py**, in a text editor (or *IDLE,* if you prefer), you can catch a glimpse of how things work. We won't go into the specifics of object-oriented programming and classes; all you need to understand is that the board we draw in *Minecraft* is an instance of the object whose description begins with the **class** at line 19. This object has its own variables and functions, and can in fact be instantiated as many times as you like. The particular instance created by the code is called **mcap** and is defined way down on line 272.

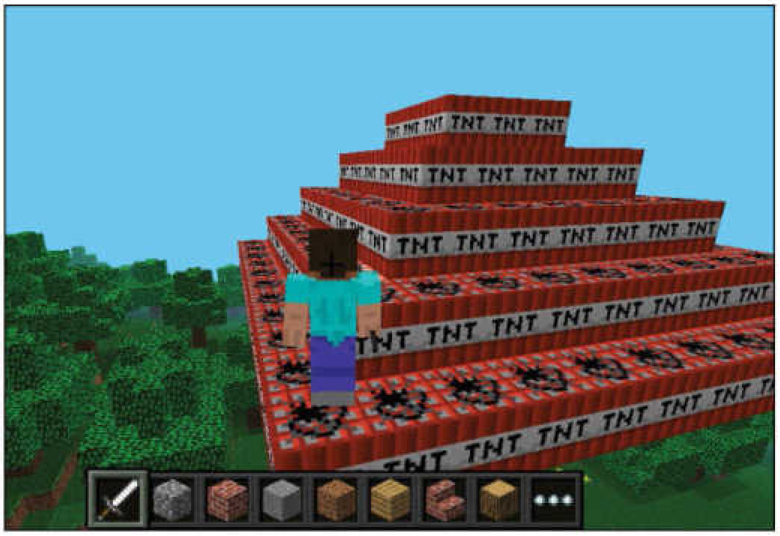
Quite early on is defined a list called **LED_WOOL_COL**, which reconciles the 16 colours of wool available with something that the LED array can understand. When we hit a virtual LED block, it changes colour and the corresponding LED on the array does likewise. This is done in the **interact()** function, which in fact deals with all the components that can be hit. Rather than using the **setBlock()** functions, the virtual Raspberry Pi is constructed of objects from the custom **shapeBlock** class (which you can explore in the **minecraftstuff.py** file). These have a property called **tag**, which is a human-readable string describing the blocks in question. When a block is hit, this property is checked using a series of **if** and **elif** (short for **else-if** – in other words, if the previous condition didn't hold, try this one) statements and the appropriate action is taken. Some of the components just display an informational message using **mc.postToChat()**; some, such as the environmental and orientation sensors, query the Astro Pi hardware and display current readings; the joystick even moves the Pi around.

Because the sensor data often has lots of decimal places that we may not be interested in, we take advantage of Python's string formatting capabilities to round things appropriately. We can use curly braces as placeholders for variables, then use the **.format()** method to determine how they are displayed. For example, the following string (from line 157) formats the readings for the combined humidity and temperature sensor rounded to two decimal places:

```
" humidity = {}, temperature = {}".format(round(humidity,2),
round(temp, 2))
```

We've already met the slightly odd **if __name__ == "__main__":** construct in our first tutorial on page 96. It's used to discern whether the program has been executed from the command line or just imported. So that the player can keep exploring, this block of code contains a main loop which constantly polls for new events (the player hitting blocks):

```
    while(True):
        #each time a block is hit pass it to the interactive
astro pi
        for blockHit in mc.events.pollBlockHits():
            mcap.interact(blockHit.pos)
        #keep reading the astro pi orientation data otherwise
it goes out of sync
        ap.get_orientation()
        #sleep for a bit
        sleep(0.1)
```



Bring TNT to life with the magic of the *Minecraft* Python API. This probably isn't a safe place to stand.

This loop runs until it is forcefully interrupted, either by pressing Ctrl+C or by killing the *Minecraft* program. We pass the position of each block-hitting event to the **interact()** function, which figures out what has happened and what to do. The main loop is wrapped in **try** block, which is a construct usually used in conjunction with an **except:** for catching errors and exceptions. In this case, it is followed with a **finally:**, which is executed (whether or not an exception arises) when the **try:** completes. In our case, this ensures that we kill our running code with Ctrl+C, the virtual Pi is dematerialised, and the Astro Pi is reset.

This is a really great project to explore and extend, so go forth and do these things. Also great work, Hannah, for coming up the idea in the first place, and mad props to Martin for allowing us to use his code. ■

Minecraft: Pi Edition API, teleporting and TNT

Even if you don't have an Astro Pi, you can still have a lot of fun with the *Minecraft* API. Once you've imported the module, you need to set up the **mc** object (which handles all communication between Python and *Minecraft),* then you're good to go. We have already seen the **postToChat()** and **setBlocks()** functions, but there are a few more. For instance, we can move the player to the coordinates of our choosing with **player.setPos()**. This means that we can write and call a simple teleport function from within the interpreter:

```
def teleport(x=0, y=0, z=0):
    mc.player.setPos(x, y, z)
```

We've given some defaults for the arguments here, so that calling **teleport()** with no arguments takes Steve back to the origin (where the Astro Pi is drawn).

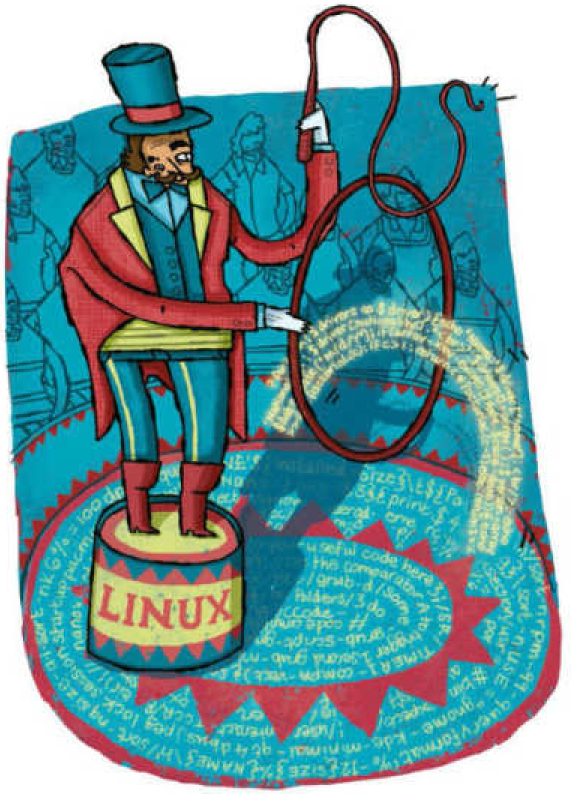
If you've had a look at the various blocks available, you'll no doubt have discovered TNT (block type 46). Sadly, there isn't a way to blow it up in-game. But fret not – mature adults can get their explosions by using some additional block data. Here's a function that'll figure out Steve's position and then assemble a substantial pyramid of live TNT above him:

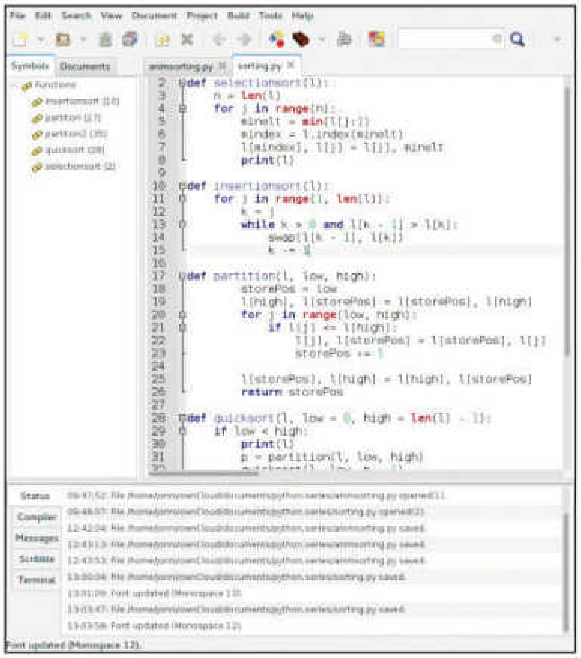
```
def tnt_pyramid():
    x, y, z = mc.player.getPos()
    for j in range(5):
        xedge_lo = x - 5 + j
        xedge_hi = xedge_lo + 10 - 2 * j
        zedge_lo = z - 5  + j
        zedge_hi = zedge_lo + 10 - 2 * j
        mc.setBlocks(xedge_lo, y+ j + 2, zedge_lo,
xedge_hi, y + j + 2, zedge_hi, 46, 1)
```

The magic is all in that final [code] 1 [/code]. This sets the TNT to be live, so that if you left-click it with your sword (or anything), then it begins to flash and pulsate ominously. At this point, you ought to migrate to a safe distance. Also, for the older, single-core Pis, the resulting chain reaction will be very taxing.

Sorting lists faster in Python

We haven't got time for this, it's time to sort out that most fundamental of algorithm classes: sorting (sort of).





```
def selectionsort(l):
    n = len(l)
    for j in range(n):
        minelt = min(l[j:])
        mindex = l.index(minelt)
        l[mindex], l[j] = l[j], minelt
        print(l)

def insertionsort(l):
    for j in range(1, len(l)):
        k = j
        while k > 0 and l[k - 1] > l[k]:
            swap(l[k - 1], l[k])
            k -= 1

def partition(l, low, high):
        storePos = low
        l[high], l[storePos] = l[storePos], l[high]
        for j in range(low, high):
            if l[j] <= l[high]:
                l[j], l[storePos] = l[storePos], l[j]
                storePos += 1

        l[storePos], l[high] = l[high], l[storePos]
        return storePos

def quicksort(l, low = 0, high = len(l) - 1):
    if low < high:
        print(l)
        p = partition(l, low, high)
```

```
09:47:52: File /home/jonny/ownCloud/documents/python series/animsorting.py opened(1).
09:48:07: File /home/jonny/ownCloud/documents/python series/sorting.py opened(2).
12:42:04: File /home/jonny/ownCloud/documents/python series/animsorting.py saved.
12:43:13: File /home/jonny/ownCloud/documents/python series/animsorting.py saved.
12:43:53: File /home/jonny/ownCloud/documents/python series/animsorting.py saved.
13:00:04: File /home/jonny/ownCloud/documents/python series/sorting.py saved.
13:01:09: Font updated (Monospace 13).
13:03:47: File /home/jonny/ownCloud/documents/python series/sorting.py saved.
13:03:58: Font updated (Monospace 12).
Font updated (Monospace 12).
```

› *Geany* is a feature-packed text editor that's great for editing Python code. Code-folding and highlighting without the bloat.

Within the human species, sorting things (putting them into some kind of order) is perceived as anywhere between a necessity (things disordered cause the most dire distress) and a dark ritual into which one should not delve. Computers, by comparison, universally prefer things to be sorted, because searching and indexing data is much easier if it is in order. Imagine how much less useful would be an unsorted telephone directory. As such, various sorting algorithms were developed early in the history of computer science. These enable a modern computer to happily sort a several thousand-strong iTunes (err, Rhythmbox) library in a matter of seconds, listing tracks with whatever ordering the user desires, be it lexicographical or chronological. In theory, one could even re-enact Rob Fleming's autobiographical ordering (the order in which he purchased them) from the Nick Hornby book *High Fidelity,* albeit several orders of magnitude faster.

Sorting may not be the most glamorous of programming topics, and indeed Python's built-in sorting methods will prove much faster than anything we program here, but it serves as a great introduction to various programming concepts. Furthermore, it enables us to show some of Python's advantages, including its no-nonsense syntax and the simplicity by which we can use graphics. To make things easier, we're going to work exclusively with lists of integers. All of the concepts apply equally well to any other kind of data, of course, but this way we have a well-defined order (ascending numerical) to aim for.

When challenged with the drudgery-filled task of, say, putting 52 cards into order, first by suit and then within each suit by face value, most people will flounder around with some haphazard piling and re-arranging, and eventually complete the task (or give up and find something better to do). Machines obviously need to be more systematic (and dedicated) in their approach. It's straightforward enough to come up with a couple of sorting algorithms that work, albeit not very quickly. We'll sum up some of these with pseudocode below. Pseudocode is a freeform way of expressing code that lies somewhere in between human language and the programming language. It's a good way to plan your programs, but don't be disheartened when the translation becomes difficult. We've actually overlooked a couple of

Quick tip

If you're using a new version of **Matplotlib** with Gnome, you might find that the animation described in the box displays only a blank window. A workaround is to add **pylab.pause(0.001)** after each **draw()** call.

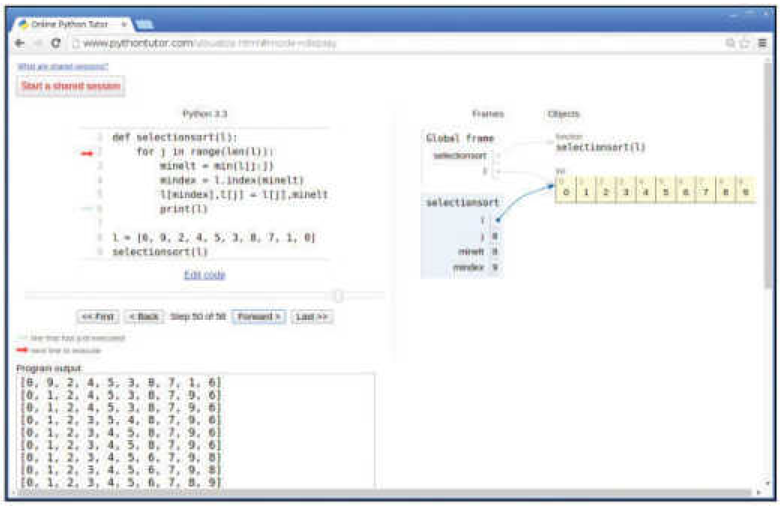
minor details here, but that will all be sorted out when we write our actual code.

```
SelectionSort(list):
  for each position j in list
    find minimum element after position j
    if minimum element is smaller than element at j
      swap them
```

Selection Sort is a fairly simple algorithm, but in simplicity lies beauty. The first loop iteration simply finds the smallest element in our list and places it, rightfully, at the beginning. We then start at the second item and see whether there is a smaller one further down the list, swapping if so. And so it goes on, until the work is done. Note that we swap elements here rather than insert them, this is so that we are only modifying two of our list items; inserting an item would require the whole list to be shunted down between the insertion and deletion points, which in the worst case would mean modifying every item in the list – a costly procedure if said list is long. We met **for** loops way back on page 50, and we can tell that **SelectionSort** will loop over items in the list at least once. But in each loop iteration we must find a minimum element, which for the first few loops (where **j** is small) involves checking most of the list items individually. We can approximate how long an algorithm takes to run by looking at these details. In **Selection Sort's** worst case we will pretty much have two loops – one nested inside the other, each going over our whole list. We use so-called big O notation to express the function's complexity as a function of the input size. In our case, we suppose our list has **n** items and the crude analysis above says that **Selection Sort** runs with complexity O(n^2). This isn't saying that Selection Sorting a list of one item will take one second (actually it takes 0 seconds) or a list of five items will take 25 seconds. What we can say is that the relationship between input size and sorting time, as **n** gets large, will be less than some constant (which depends on the machine) times n^2. This is slightly disappointing news – in layman's terms, **Selection Sort** gets pretty slow pretty quickly.

Here is another naive sorting algorithm, which is called **Insertion Sort**:

```
InsertSort(list):
  for each position j in list
    k = j
    while number at position k is less then that at position k - 1
      swap numbers at positions k and k - 1
      k = k - 1
```



```
def selectionsort(l):
    for j in range(len(l)):
        minelt = min(l[j:])
        mindex = l.index(minelt)
        l[mindex],l[j] = l[j],minelt
        print(l)

l = [6, 9, 2, 4, 5, 3, 8, 7, 1, 0]
selectionsort(l)
```

```
[0, 9, 2, 4, 5, 3, 8, 7, 1, 6]
[0, 1, 2, 4, 5, 3, 8, 7, 9, 6]
[0, 1, 2, 4, 5, 3, 8, 7, 9, 6]
[0, 1, 2, 3, 5, 4, 8, 7, 9, 6]
[0, 1, 2, 3, 4, 5, 8, 7, 9, 6]
[0, 1, 2, 3, 4, 5, 8, 7, 9, 6]
[0, 1, 2, 3, 4, 5, 6, 7, 9, 8]
[0, 1, 2, 3, 4, 5, 6, 7, 9, 8]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

› If you're looking for an excellent resource for learning and debugging your code, visit http://pythontutor.com. Here we can see Selection Sort in action.

This time we are upfront about our two nested loops, so again we can deduce O(n^2) complexity. We swap items that are not in order, and then move backwards down the list continuing to swap out-of-order entries. Once we get back to the beginning, we resume the outer loop, which proceeds towards the end. What is not immediately obvious, but is nonetheless interesting, is that this algorithm (like **Selection Sort**) has sorted the first **i** entries after that many iterations of the outer loop. As it proceeds, an out-of-order element may be swapped in, but by the end of the inner loop it will be in its right place, having trickled down there by a lengthy swap sequence. Time complexity notwithstanding, **InsertionSort**, in many situations, outperforms **Selection Sort**, the most obvious one being where the list is nearly sorted to begin with – in other words, list items are not too far from their rightful places. In this situation, the inner **while** loop terminates quickly, so the algorithm actually behaves with apparently linear, O(n), complexity.

There are a few other naive sorting algorithms that may be worthy of your perusal, including **Bubble Sort** (aka **Sinking Sort**) and its parallel version **Odd-Even Sort**, but we have much to get through so it's time to look at a grown-up method called **Quicksort**. **Quicksort** was developed in 1959 and, thanks to its swift O(n.log n) complexity, was quickly adopted as the default sorting algorithm in Unix and other systems. It is markedly more involved than the algorithms

»

Quick tip

Very often, the simple algorithms outperform **Quicksort** for small lists. As such, you can create a hybrid sorting algorithm which uses **Quicksort** initially, but as the list is divided, reverts to **Selection Sort**, for example.

Visualising sorting algorithms

Python has some incredibly powerful modules that can help visualise the sorting algorithms discussed here. We're going to use the **Matplotlib** package, which can do all manner of plotting and graphing. It can be found in the **python3-matplotlib** package in Debian-based distros. There isn't room here to even scratch the surface of what's possible with **Matplotlib**, but we can show that with just a couple of extra lines, our sorting algorithms can be brought to life. Here we've modified our **insertionsort()** function, and added some boilerplate code to set everything up. You are encouraged to modify **selectionsort()** and **quicksort()** too – just add the **line.set_ydata()** and **pylab.draw()** calls after each line that modifies the list.

```
#! /usr/bin/env python3
import pylab
from random import shuffle

def insertionsort_anim(l):
  for j in range(1, len(l)):
    k = j
    while k > 0 and l[k - 1] > l[k]:
      l[k - 1], l[k] = l[k], l[k - 1]
      line.set_ydata(a)
      pylab.draw()
      k -= 1

pylab.ion() # turn on interactive mode
a = list(range(300))
x = list(range(len(a)))
shuffle(a)
line, = pylab.plot(x,a,'m.',markersize=6)
insertionsort_anim(a)
```

Save this file as **~/animsorting.py** and then do **chmod +x ~/animsorting.py** so that it can be run from the comfort of the command line. It's not the fastest way to display graphics (all the data is redrawn after each list update), but it's easy, and much more exciting than the **print** statements we used earlier.

» hitherto explored, but the rewards shall prove worthy of your attention. If **Quicksort** had a motto, it would be "Impera et divide" because at each stage it does some sneaky rearranging and then divides the list into two sublists and acts on them in isolation. (**Mergesort**, another algorithm, does the dividing first and the conquering after.) The list is rearranged and divided according to a pivot value – all items less than the pivot are moved to its left (nearer the beginning of the list), and all items greater than the pivot are moved to its right. This stage is called the partition operation; once it is complete, we work on the two sublists to the left and right of the pivot. And here begins the real fun, because we apply exactly the same methodology to our two smaller lists. Eventually, the division leads to sublists of size 0 or 1, which by definition are sorted.

Quicksort is, then, an example of a recursive algorithm – an algorithm that calls itself. This is a particularly tricky subject for the novice programmer to get their head around, mostly because the tendency is to imagine some kind of infinite chain of function calls, spiralling into the depths of oblivion and memory exhaustion. While it's entirely possible to program such a monster, a decent recursive algorithm ought to be good at how (and when) it calls itself. Typically, our recursive function calls involve smaller arguments (for example, a sublist rather than the whole list) and there are so-called base cases where recursion is not used. For **Quicksort**, the base cases are the aforementioned lists of length 0 or 1. Neglecting the details of the partition operation for a moment, our **Quicksort** pseudocode looks deceptively simple (excepting perhaps the recursion element). It takes, besides an unsorted list, two extra parameters (low and high). These specify the low and high indices between which we should sort. For the initial function call, these are zero and the length of the list, and for subsequent calls they are either from the low element to the pivot element or from the pivot element to the high element.

```
Quicksort(list, low, high):
    If low < high:
        p = Partition(list, low, high)
        Quicksort(list, low, p)
        Quicksort(list, p + 1, high)
```

The first thing that the **Partition()** operation must do is choose a pivot. Unfortunately, there is no straightforward way to do this – ideally one would choose a value that is going to end up about halfway in the sorted list, but this cannot be achieved a priori. In practice, you can often get away with choosing a random value here, or if a more systematic approach is preferred, the median of the first, middle and final elements in the list. We'll leave the pivot-choosing as a black box function **choosePivot()**, which returns the index of the pivot. So our partition operation, which also returns the index of the pivot, looks like:

```
Partition(list, low, high):
    pivotPos = choosePivot(list, low, high)
    pivotVal = list[pivotPos]
    # put the pivot at the end of the list
    Swap list[pivotPos] and list[high]
    storePos = low
    For each position j from low to high - 1:
        If list[j] <= pivotVal:
            Swap  list[j] and list[storePos]
            storePos = storePos + 1
    # put the pivot after all the lower entries
    Swap list[storePos] and list[high]
    Return storePos
```

That may seem complicated, but bear in mind all it's doing is arranging the list so that items less than the pivot are to its left (not necessarily in order). We have only one loop here, so the partition operation runs in O(n) time. In an ideal situation, the partition operation divides the list into two roughly equal parts, so at each stage of the recursion we half the list size. This is where the log(n) in the complexity comes from, since halving a list of size n about log-to-the-base-two of n times results in a singleton list, whereupon the recursion ceases. It is possible for **Quicksort** to perform much slower than stated, though – for example, if we start with an already sorted list and always use the last (greatest) element as a pivot, we are reduced to O(n^2) complexity.

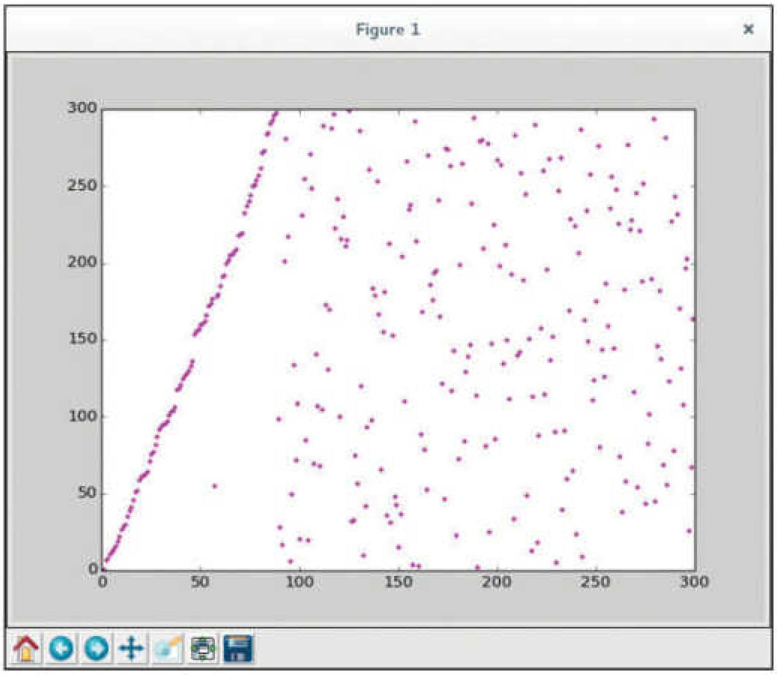
Coding it up

It's time to translate our three algorithms into Python. We saw briefly how to define functions in the prequel, start the block with a **def** keyword, and then indent all the code that belongs to that function. In our pseudocode we referred to our list as, rather unimaginatively, **list**. Unfortunately, that's one of not terribly many reserved words in Python, so we'll instead use **l** (for llama). **Selection sort** then looks like:

```
def selectionsort(l):
    for j in range(len(l)):
        minelt = min(l[j :])
        mindex = l.index(minelt)
        if minelt < l[j]:
            l[mindex] = l[j]
            l[j] = minelt
```

For simplicity, we're doing an inplace sort here – we've modified the original list so we don't need to return anything. The Python code is quite different from the pseudocode, but much of this is cosmetic. We use slicing to find the minimum element on or after position **j** and then use the **index()** method to find that minimum's position in the list. The **if** statement is actually superfluous here (we've left it in to better match the pseudocode), because if the condition is false (which would mean the minimum element was at position **j**), then it would harmlessly swap a list item with itself. There's no explicit **swap** operation, so we have to do that manually in the last two lines. Incidentally, these can be further condensed to a single

Elements undergoing Insertion Sort trickle one at a time from right to left. Watch it at night-time as an alternative to counting sheep.



Optional arguments and speed considerations

Because our **Quicksort()** function needs the **low** and **high** parameters, the initial call looks like **quicksort(l, 0, len(a) - 1)**. Python does allow default values for arguments to be specified – you simply suffix the argument with an **=**. However, we can't use, for example, **high = len(l)** because we're not allowed to use internal variables. The workaround would be to make the first part of the function look like:

```
def quicksort(l, low = 0, high = None):
    if high is None:
        high = len(l)
```

You need some reasonably-sized lists to see **Quicksort's** benefits – we discovered that it could sort 10,000 items in about 0.3 seconds, whereas the others took much longer, particularly **Insertion Sort**, which took about 14 seconds. Exact timing depends on the particular list, of course, but it's also worth noting that (much) faster implementations of these algorithms are possible. Lists in Python have their own **.sort()** method, which happily sort about 50,000 items instantly.

We've tried to use code here that's at once simple and as close to the given pseudocode as possible. In general, Python isn't a language known for its speed, unfortunately, particularly when working with lists, though many of its other functions and modules are implemented through its C API.

```
l[mindex],l[j] = l[j],l[mindex]
```

This means that the whole algorithm is done in four lines. Swapping variables without this neat Pythonic trick would involve setting up a temporary variable to hold one of the values. Save this file as **~/sorting.py** or similar, then start the Python 3 interpreter in your home directory. You should be able to import your module like so:

```
>>> import('sorting')
```

Assuming you didn't see any error messages or make any typos, you can then apply your algorithm to a shuffled list. We'll use the **random** module to shuffle a list from 0 to 9 and test this:

```
>>> import random
>>> l = list(range(10))
>>> random.shuffle(l)
>>> sorting.selectionsort(l)
>>> l
```

Voilà – a freshly sorted list. Of course, it's nice to watch the algorithm progress. So add a **print(l)** statement at the end of the loop, with the same indentation so it is still within said loop. To reload the module and apply the changes, we need to use the importlib module's **reload()** function, the standard **import** statement won't notice changes on already loaded modules. Or you can just exit (Ctrl+D) and restart the interpreter. The screenshot on page 103 shows the output for the list [6, 9, 2, 4, 5, 3, 8, 7, 1, 0]. We can see the list becoming sorted from left to right, and that the last stage of the loop doesn't do anything.

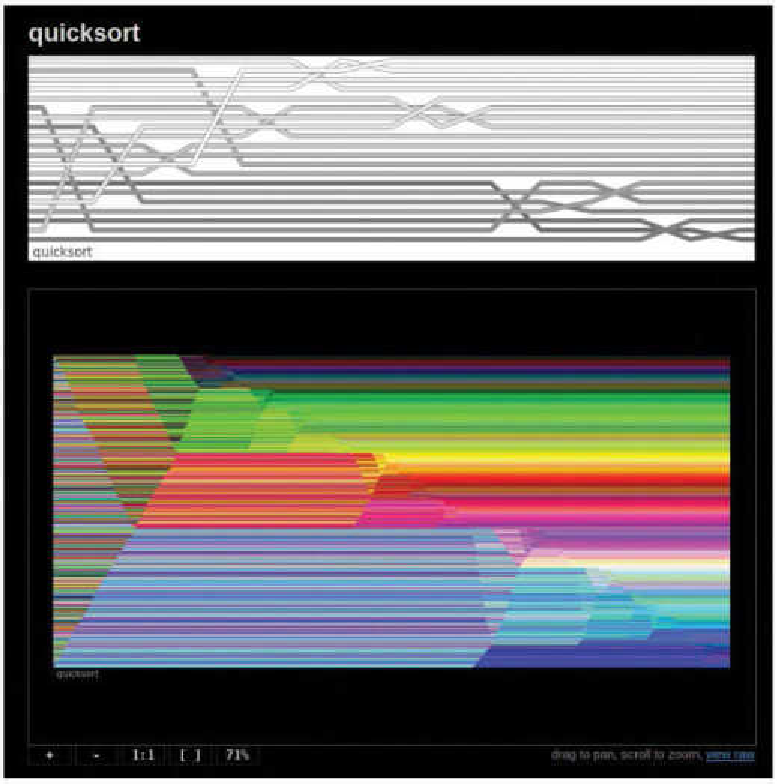
Moving on to **Insertion Sort**, we translate the pseudocode into Python code, which does not look drastically different. Add this to the **sorting.py** file.

```
def insertionsort(l):
  for j in range(1, len(l)):
    k = j
    while k > 0 and l[k - 1] > l[k]:
      l[k - 1], l[k] = l[k], l[k - 1]
      k -= 1
```

We start the outer loop at **1** so that the **l[k-1]** comparison in the **while** statement doesn't break things. We also add **k > 0** to the **while** condition because the loop should stop before **k** gets to zero, rather than try (and fail) to retreat past the beginning of the list. We've used our swapping shortcut from before and in the final line use the **-=** notation, which is short for **k = k - 1**.

Again, add **print** statements to either the **for** or **while** loops to see the algorithms progress. Smaller elements are swapped to the beginning of the list one place at a time until the list is sorted.

The **Quicksort** implementation is a bit more involved. For the partition operation, we're going to take the bold step of just using the first element for the pivot, so we don't need to worry about the first few lines of pseudocode.



```
def partition(l, low, high):
    storePos = low
    l[high], l[storePos] = l[storePos], l[high]
    for j in range(low, high):
      if l[j] <= l[high]:
        l[j], l[storePos] = l[storePos], l[j]
        storePos += 1
    l[storePos], l[high] = l[high], l[storePos]
    return storePos

def quicksort(l, low, high):
  if low < high:
    p = partition(l, low, high)
    quicksort(l, low, p - 1)
    quicksort(l, p + 1, high)
```

Try visualising this using the guide in the box on page 103 – the **pylab.draw()** function should be called after the final swap in the **partition()** function, so that the pivot element is plotted in the correct place.

There are many more sorting algorithms, and research is ongoing to find more, but hopefully this introduction has shown you something of sorting as well as increased your Python skills. We have more fun lessons ahead. ■

› Besides animations, sorting algorithms can be visualised as wave diagrams. Find out more at http://sortvis.org.

THE EASY WAY TO LEARN WINDOWS

WINDOWS 10
LEARN HOW TO MASTER THE MEDIA PLAYER APP

WINDOWS 8.1
OPEN FILES IN YOUR FAVOURITE PROGRAMS

TIPS & TRICKS
HOW TO EXTEND YOUR LAPTOP'S BATTERY LIFE

Windows Help & Advice

GET STARTED WITH WINDOWS 10

The complete guide to your new PC!

- ☑ Master the Desktop and Start menu
- ☑ Search smarter with Cortana
- ☑ Discover the very best apps!

50 PAGES OF STEP-BY-STEP WINDOWS GUIDES!

BEGINNERS' GUIDE

How to keep your information private in Windows 10

PC HELP

WINDOWS TIPS, TRICKS & ADVICE

- Top 10 New Year's PC resolutions
- Scan and archive old photos
- Log in with facial recognition

Turn to p39 now!

Future

FEBRUARY 2016

100% JARGON FREE

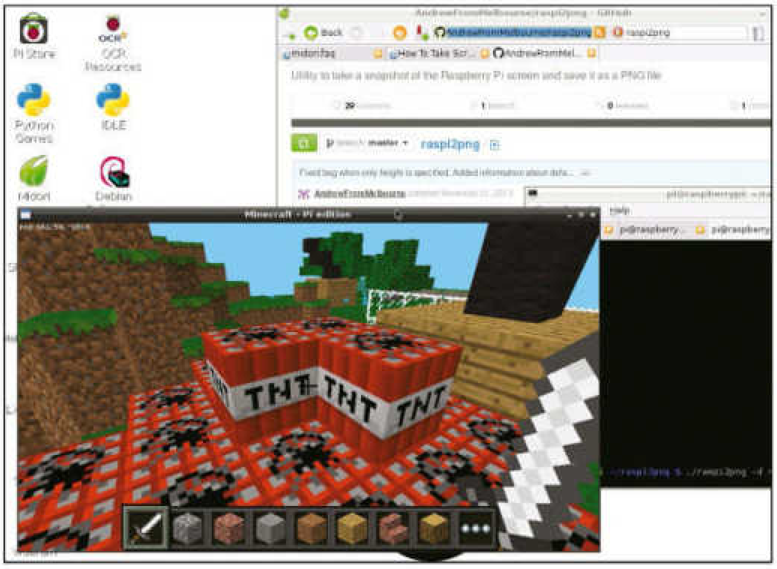
AVAILABLE IN STORE AND ONLINE
www.myfavouritemagazines.co.uk

Minecraft: Start hacking

Use Python on your Pi to merrily meddle with Minecraft.

Arguably more fun than the generously provided *Wolfram Mathematica: Pi Edition* is Mojang's generously provided *Minecraft: Pi Edition*. The latter is a cut-down version of the popular *Pocket Edition*, and as such lacks any kind of life-threatening gameplay, but includes more blocks than you can shake a stick at, and three types of saplings from which said sticks can be harvested.

This means that there's plenty of stuff with which to unleash your creativity, then, but all that clicking is hard work, and by dint of the edition including of an elegant Python API, you can bring to fruition blocky versions of your wildest dreams with just a few lines of code.



Don't try this at home, kids... actually *do* try this at home.

Assuming you've got your Pi up and running, the first step is downloading the latest version from **http://pi.minecraft.net** to your home directory. The authors stipulate the use of Raspbian, so that's what we'd recommend – your mileage may vary with other distributions. *Minecraft* requires the X server to be running so if you're a boot-to-console type you'll have to **startx**. Start *LXTerminal* and extract and run the contents of the archive like so:

```
$ tar -xvzf minecraft-pi-0.1.1.tar.gz
$ cd mcpi
$ ./minecraft-pi
```

See how smoothly it runs? Towards the top-left corner you can see your x, y and z co-ordinates, which will change as you navigate the block-tastic environment. The x and z axes run parallel to the floor, whereas the y dimension denotes altitude. Each block (or voxel, to use the correct parlance) which makes up the landscape is described by integer co-ordinates and a BlockType. The 'floor' doesn't really have any depth, so is, instead, said to be made of tiles. Empty space has the BlockType AIR, and there are about 90 other more tangible substances, including such delights as GLOWING_OBSIDIAN and TNT. Your player's co-ordinates, in contrast to those of the blocks, have a decimal part since you're able to move continuously within AIR blocks.

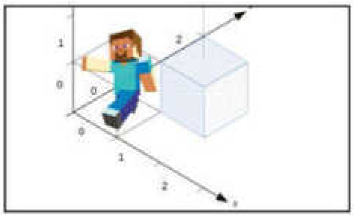
The API enables you to connect to a running *Minecraft* instance and manipulate the player and terrain as befits your megalomaniacal tendencies. In order to service these our first task is to copy the provided library so that we don't mess with the vanilla installation of *Minecraft*. We'll make a special folder for all our mess called **~/picraft**, and put all the API stuff in **~/picraft/minecraft**. Open *LXTerminal* and issue the following directives:

```
$ mkdir ~/picraft
$ cp -r ~/mcpi/api/python/mcpi ~/picraft/minecraft
```

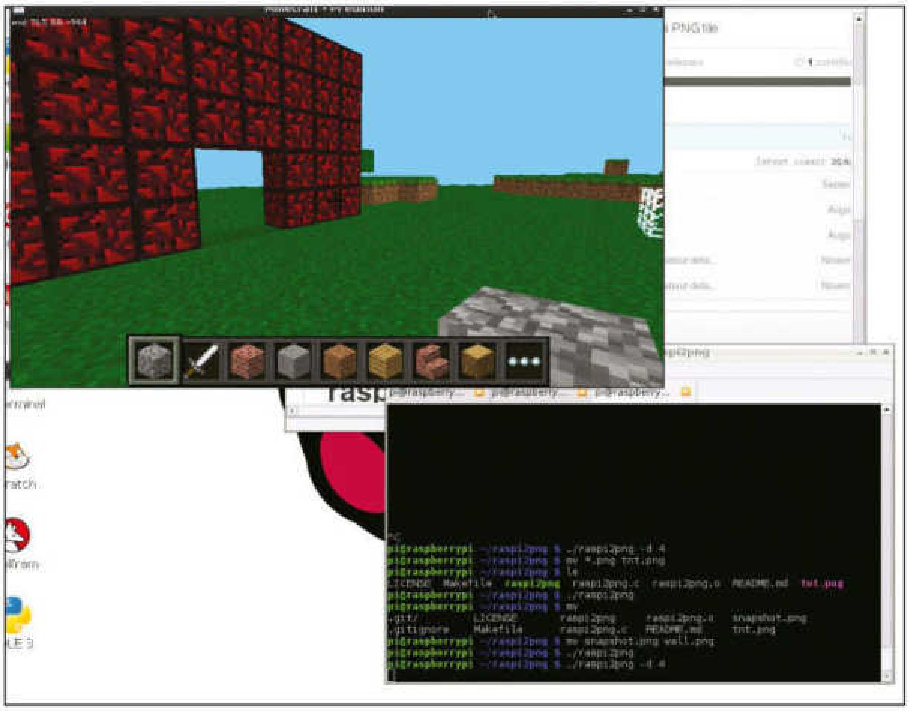
Dude, where's my Steve?

Here we can see our intrepid character (Steve) inside the block at (0,0,0). He can move around inside that block, and a few steps in the x and z directions will take Steve to the shaded blue block. On this rather short journey he will be in more than one block at times, but the *Minecraft* API's **getTilePos()** function will choose the block which contains most of him.

Subtleties arise when trying to translate standard concepts, such as lines and polygons from Euclidean space into discrete blocks. A 2D version of this problem occurs whenever you render any kind of vector graphics: Say, for instance, you want to draw a line between two points on the screen, then unless the line is horizontal or vertical, a decision has to be made as to which pixels need to be coloured in. The earliest solution to this was provided by Jack Elton Bresenham in 1965, and we will generalise this classic algorithm to three dimensions a little later in this chapter.



Isometric projection makes *Minecraft*-world fit on this page.



```
^C
pi@raspberrypi ~/raspi2png $ ./raspi2png -d 4
pi@raspberrypi ~/raspi2png $ mv *.png tnt.png
pi@raspberrypi ~/raspi2png $ ls
LICENSE  Makefile  raspi2png  raspi2png.c  raspi2png.o  README.md  tnt.png
pi@raspberrypi ~/raspi2png $ ./raspi2png
pi@raspberrypi ~/raspi2png $ ls
.git/        LICENSE      raspi2png      raspi2png.o   snapshot.png
.gitignore   Makefile     raspi2png.c    README.md     tnt.png
pi@raspberrypi ~/raspi2png $ mv snapshot.png wall.png
pi@raspberrypi ~/raspi2png $ ./raspi2png
pi@raspberrypi ~/raspi2png $ ./raspi2png -d 4
```

All manner of improbable structures can be yours.

Now without further ado, let's make our first *Minecraft*ian modifications. We'll start by running an interactive Python session alongside *Minecraft*, so open another tab in *LXTerminal*, start *Minecraft* and enter a world then Alt-Tab back to the terminal and open up Python in the other tab. Do the following in the Python tab:

```
import minecraft.minecraft as minecraft
import minecraft.block as block
mc = minecraft.Minecraft.create()
posVec = mc.player.getTilePos()
x = posVec.x
y = posVec.y
z = posVec.z
mc.postToChat(str(x)+' '+ str(y) +' '+ str(z))
```

Behold, our location is emblazoned on the screen for a few moments (if not, you've made a mistake). These co-ordinates refer to the current block that your character occupies, and so have no decimal point. Comparing these with the co-ordinates at the top-left, you will see that these are just the result of rounding down those decimals to integers (e.g. -1.1 is rounded down to -2). Your character's co-ordinates are available via **mc.player.getPos()**, so in some ways **getTilePos()** is superfluous, but it saves three float to int coercions so we may as well use it. The API has a nice class called Vec3 for dealing with three-dimensional vectors, such as our player's position. It includes all the standard vector operations such as addition and scalar multiplication, as well as some other more exotic stuff that will help us later on.

We can also get data on what our character is standing on. Go back to your Python session and type:

```
curBlock = mc.getBlock(x, y - 1, z)
mc.postToChat(curBlock)
```

Here, **getBlock()** returns an integer specifying the block type: 0 refers to air, 1 to stone, 2 to grass, and you can find all the other block types in the file block.py in the **~/picraft/** minecraft folder we created earlier. We subtract 1 from the y value since we are interested in what's going on underfoot – calling **getBlock()** on our current location should always return 0, since otherwise we would be embedded inside something solid or drowning.

As usual, running things in the Python interpreter is great for playing around, but the grown up way to do things is to put all your code into a file. Create the file **~/picraft/gps.py** with the following code.

```
import minecraft.minecraft as minecraft
import minecraft.block as block
mc = minecraft.Minecraft.create()
oldPos = minecraft.Vec3()
while True:
    playerTilePos = mc.player.getTilePos()
    if playerTilePos != oldPos:
        oldPos = playerTilePos
        x = playerTilePos.x
        y = playerTilePos.y
        z = playerTilePos.z
        t = mc.getBlock(x, y – 1, z)
        mc.postToChat(str(x) + ' ' + str(y) + '  ' + str(z) + ' ' +
str(t))
```

Now fire up *Minecraft*, enter a world, then open up a terminal and run your program:

```
$ python gps.py
```

The result should be that your co-ordinates and the BlockType of what you're stood on are displayed as you move about. Once you've memorized all the BlockTypes (joke), Ctrl+C the Python program to quit.

We have covered some of the 'passive' options of the API, but these are only any fun when used in conjunction with the more constructive (or destructive) options. Before we sign off, we'll cover a couple of these. As before start *Minecraft* and a Python session, import the *Minecraft* and block modules, and set up the mc object:

```
posVec = mc.player.getTilePos()
x = posVec.x
y = posVec.y
z = posVec.z
for j in range(5):
    for k in range(x - 5, x + 5)
        mc.setBlock(k, j, z + 1, 246)
```

Behold! A 10x5 wall of glowing obsidian has been erected adjacent to your current location. We can also destroy blocks by turning them into air. So we can make a tiny tunnel in our obsidian wall like so:

```
mc.setBlock(x, y, z + 1, 0)
```

Assuming of course that you didn't move since inputting the previous code.

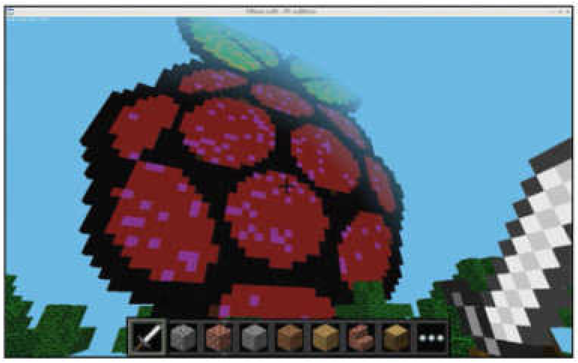
In the rest of this chapter, we'll see how to build and destroy some serious structures, dabble with physics, rewrite some of the laws thereof, and go a bit crazy within the confines of our 256x256x256 world. Until then, try playing with the **mc.player.setPos()** function. Teleporting is fun! ■

Quick tip

Check out Martin O'Hanlon's website **www.stuffaboutcode.com**, which includes some great examples of just what the API is capable of.

Minecraft: Image wall importing

Have you ever wanted to reduce your pictures to 16 colour blocks? You haven't? Tough – we're going to tell you how regardless.



Not some sort of bloodshot cloud, but a giant raspberry floating in the sky. Just another day at the office.

Technology has spoiled us with 32-bit colour, multi-megapixel imagery. Remember all those blocky sprites from days of yore, when one had to invoke something called one's imagination in order to visualise what those giant pixels represented? In this tutorial we hark back to those halcyon days from the comfort of *Minecraft*-world, as we show you how to import and display graphics using blocks of coloured wool. Also Python. And the Raspberry Pi.

The most colourful blocks in *Minecraft* are wool (blockType 35): there are 16 different colours available, which are selected using the **blockData** parameter. For this tutorial we shall use these exclusively, but you could further develop things to use some other blocks to add different colours to your palette. The process of reducing an image's palette is an example of quantization – information is removed from the image and it becomes smaller. In order to perform this colour quantization we first need to define our new restrictive palette, which involves specifying the Red, Green and Blue components for each of the 16 wool colours. This would be a tedious process, involving importing an image of each wool colour into *Gimp* and using the colour picker tool to obtain the component averages, but fortunately someone has done all the hard work already.

Standard setup

If you've used *Minecraft: Pi Edition* before you'll be familiar with the drill, but if not this is how to install *Minecraft* and copy the API for use in your code.

We're going to assume you're using Raspbian, and that everything is up to date. You can download *Minecraft* from **http://pi.minecraft.net**, then open a terminal and unzip the file as follows (assuming you downloaded it to your home directory):

```
$ tar -xvzf ~/minecraft-pi-0.1.1.tar.gz
```

All the files will be in a subdirectory called **mcpi**. To run *Minecraft* you need to have first started X, then from a terminal do:

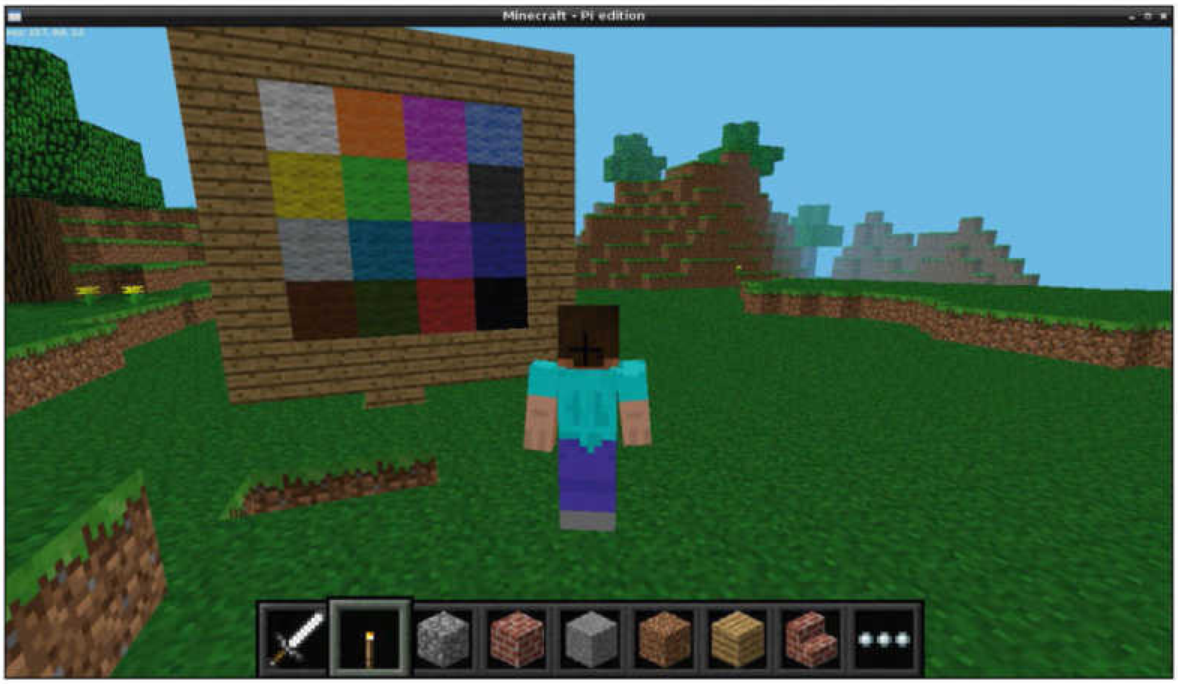
```
$ cd ~/mcpi
$ ./minecraft-pi
```

It is a good idea to set up a working directory for your *Minecraft* project, and to copy the API there. The archive on the disk will extract into a directory called **mcimg**, so you can extract it to your home directory and then copy the api files in the following way:

```
$ tar -xvzf mcimg.tar.gz
$ cp -r ~/mcpi/api/python/mcpi ~/mcimg/minecraft
```

For this tutorial we're going to use the PIL (Python Imaging Library), which is old and deprecated but is more than adequate for this project's simple requirements. It can import your .jpg and .png files, among others, so there's no need to fiddle around converting images. Install it as follows:

```
$ sudo apt-get install python-imaging
```



With just 16 colours, Steve can draw anything he wants (inaccurately).

We also need to resize our image – *Minecraft*-world is only 256 blocks in each dimension, so since we will convert one pixel to one block our image must be at most 256 pixels in its largest dimension. However, you might not want your image taking up all that space, and blocks cannot be stacked more than 64 high, so the provided code resizes your image to 64 pixels in the largest dimension, maintaining the original aspect ratio. You can modify the **maxsize** variable to change this behaviour, but the resultant image will be missing its top if it is too tall.

The PIL module handles the quantization and resizing with one-line simplicity, but we must first define the palette and compute the new image size. The palette is given as a list of RGB values, which we then pad out with zeroes so that it is of the required 8-bit order. For convenience, we will list our colours in order of the **blockData** parameter.

```
mcPalette = [
    221,221,221, # White
    219,125,62,  # Orange
    179,80,188,  # Magenta
    107,138,201, # Light Blue
    177,166,39,  # Yellow
    65,174,56,   # Lime Green
    208,132,153, # Pink
    64,64,64,    # Dark Grey
    154,161,161, # Light Grey
    46,110,137,  # Cyan
    126,61,181,  # Purple
    46,56,141,   # Blue
    79,50,31,    # Brown
    53,70,27,    # Green
    150,52,48,   # Red
    25,22,22,    # Black
]

mcPalette.extend((0,0,0) * 256 - len(mcPalette) / 3)
```

Unfortunately the “/ 3” is missing from the code at **http://bit.ly/ca2015ref** though it is a mistake without any real consequence (phew). Padding out the palette in this manner does however have the possibly unwanted side-effect of removing any really black pixels from your image. This happens because their value is closer to absolute black (with which we artificially extended the palette) than the very slightly lighter colour of the ‘black’ wool. To work around this you can change the **(0,0,0)** above to **(25,22,22)**, so that there are no longer any absolute blacks to match against. A reasonable hack if you’re working with a transparent image is to replace this value with your image’s background colour, then the transparent parts will not get drawn. We make a new single-pixel dummy image to hold this palette:

```
mcImagePal = Image.new("P", (1,1))
mcImagePal.putpalette(mcPalette)
```

The provided archive includes the file test.png, which is in fact the *Scratch* mascot, but you are encouraged to replace this line with your own images to see how they survive the {res,quant}ize. You can always TNT the bejesus out of them if you are not happy. To ensure the aspect ratio is accurate we use a **float** in the division to avoid rounding to an integer.

```
mcImage = Image.open("test.png")
width = mcImage.size[0]
height = mcImage.size[1]
ratio = height / float(width)
maxsize = 64
```

As previously mentioned, blocks in *Minecraft*-world do not stack more than 64 high (perhaps for safety reasons). The next codeblock proportionally resizes the image to 64 pixels in its largest dimension.

```
if width > height:
    rwidth = maxsize
    rheight = int(rwidth * ratio)
else:
    rheight = maxsize
    rwidth = int(rheight / ratio)
```

If you have an image that is much longer than it is high, » then you may want to use more than 64 pixels in the horizontal dimension and fix the height at 64. Replacing the above block with just the two lines of the **else** clause would achieve precisely this.

Now we convert our image to the RGB colourspace, so as not to confuse the **quantize()** method with transparency information, and then force upon it our woollen palette and new dimensions. You might get better results by doing the resize first and the quantization last, but we prefer to keep our operations in lexicographical order.

```
mcImage = mcImage.convert("RGB")
mcImage = mcImage.quantize(palette = mcImagePal)
mcImage = mcImage.resize((rwidth,rheight))
```

For simplicity, we will position our image close to Steve's location, five blocks away and aligned in the x direction to be precise. If Steve is close to the positive x edge of the world, or if he is high on a hill, then parts of the image will sadly be lost. Getting Steve's coordinates is a simple task:

```
playerPos = mc.player.getPos()
x = playerPos.x
y = playerPos.y
z = playerPos.z
```

Then it is a simple question of looping over both dimensions of the new image, using the slow but trusty **getpixel()** method, to obtain an index into our palette, and using the **setBlocks()** function to draw the appropriate colour at the appropriate place.

If your image has an alpha channel then *getpixel()* will return None for the transparent pixels and no block will be drawn. To change this behaviour one could add an **else** clause to draw a default background colour. Image co-ordinates start with (0,0) in the top-left corner, so to avoid drawing upside-down we subtract the iterating variable **k** from **rheight**.

```
for j in range(rwidth):
    for k in range(rheight):
        pixel = mcImage.getpixel((j,k))
        if pixel < 16:
            mc.setBlock(j + x + 5, rheight - k + y, z, 35, pixel)
```

To do all the magic, start *Minecraft* and move Steve to a position that befits your intended image. Then open a terminal and run:

```
$ cd ~/mcimg
$ python mcimg.py
```

So that covers the code, but you can have a lot of fun by expanding on this idea. A good start is probably to put the contents of **mcimg.py** into a function. You might want to give this function some arguments too. Something like the following could be useful as it enables you to specify the image file and the desired co-ordinates:

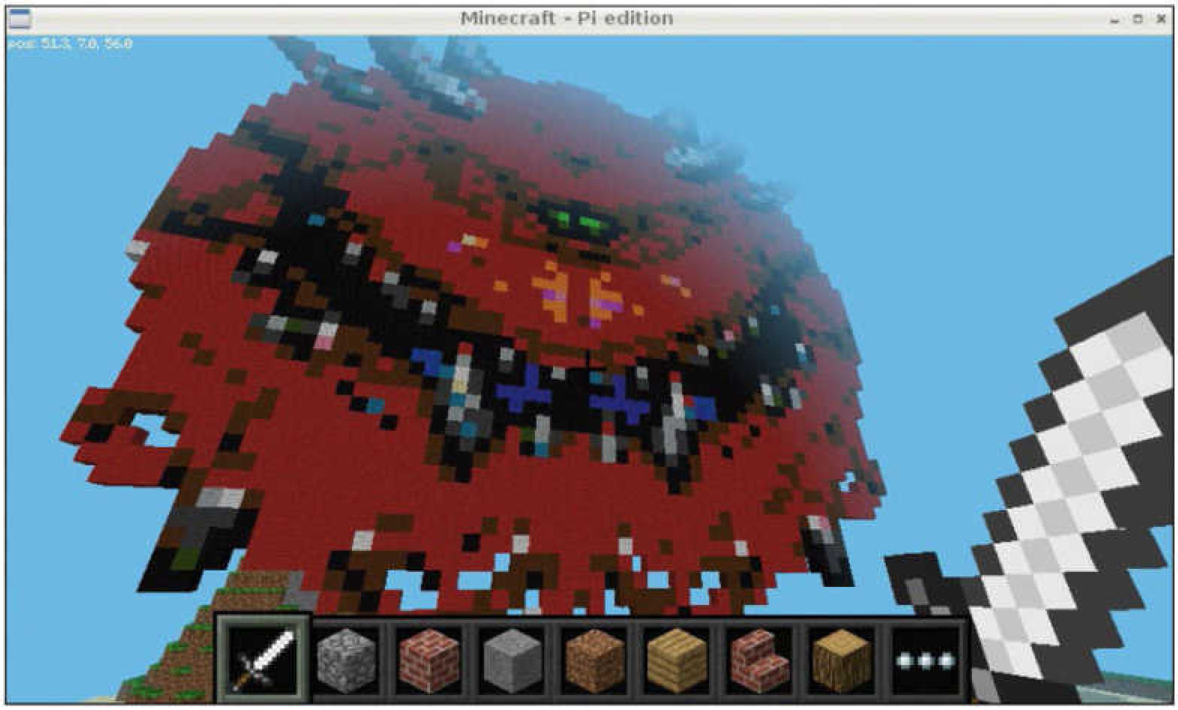
```
def drawImage(imgfile, x=None, y=None, z=None):
    if x == None:
        playerPos = mc.player.getPos()
        x = playerPos.x
        y = playerPos.y
        z = playerPos.z
```

If no co-ordinates are specified, then the player's position is used. If you have a slight tendency towards destruction, then you can use live TNT for the red pixels in your image. Just replace the **mc.setBlock** line inside the drawing loop with the following block:

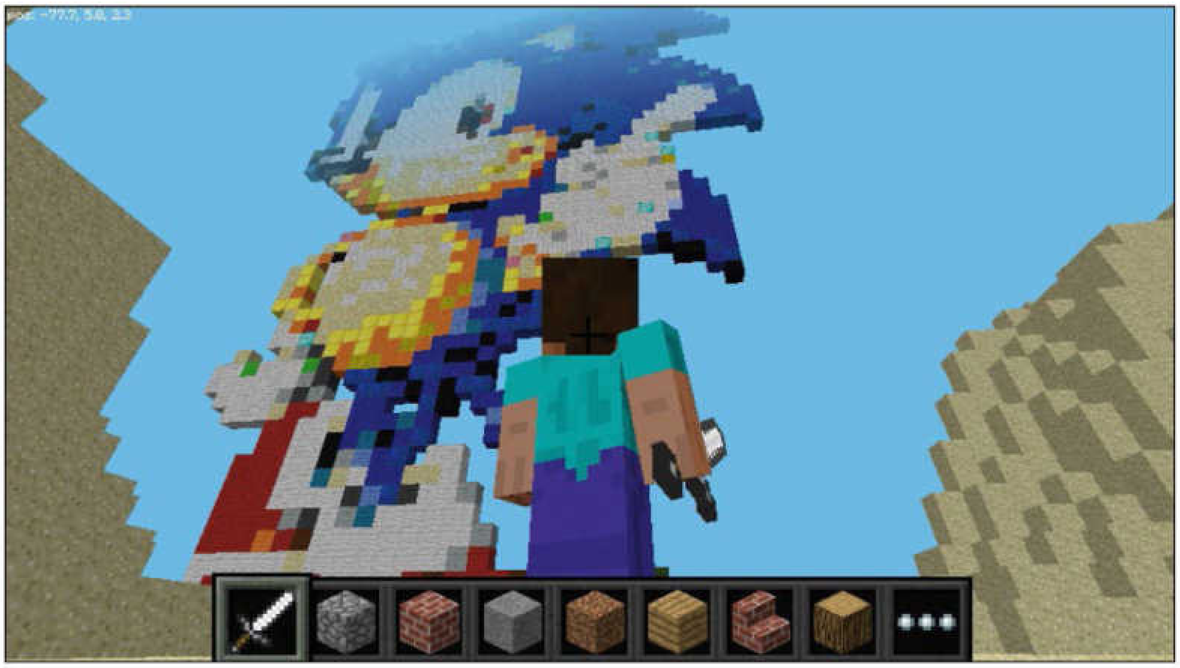
```
if pixel == 14:
    mc.setBlock(j + x + 5, rheight - k + y, z, 46, 1)
else:
    mc.setBlock(j + x + 5, rheight - k + y, z,
mcPaletteBlocks[pixel])
```

If you don't like the resulting image, then it's good news, everyone – it is highly unstable and a few careful clicks on the TNT blocks will either make some holes in it or reduce it to dust. It depends how red your original image was.

While *Minecraft* proper has a whole bunch of colourful blocks, including five different types of wooden planks and



Unlike in *Doom*, this strawberry/ cacodemon doesn't spit fireballs at you. This is good.



› That's right, Steve, go for the ankles. Let's see how fast he runs without those!

stairs, six kinds of stone, emerald, and 16 colours of stained glass, the Pi Edition is a little more restrictive. There are some good candidates for augmenting your palette, though:

Blockname	Block ID	Red	Green	Blue
Gold	41	241	234	81
Lapis Lazuli	22	36	61	126
Sandstone	24	209	201	152
Ice	79	118	165	244
Diamond	57	116	217	212

We have hitherto had it easy insofar as the **mcPalette** index aligned nicely with the coloured wool blockData parameter. Now that we're incorporating different blockTypes things are more complicated, so we need a lookup table to do the conversion. Assuming we just tack these colours on to the end of our existing **mcPalette** definition, like so:

```
mcPalette = [ …
   241,234,81,
   36,61,126,
   209,201,152,
   118,165,244,
   116,217,212
]
mcPaletteLength = len(mcPalette / 3)
```

then we can structure our lookup table as follows:

```
mcLookup = []
for j in range(16):
   mcLookup.append((35,j))
mcLookup += [(41,0),(22,0),(24,0),(79,0),(57,0)]

```

Thus the list **mcLookup** comprises the blockType and blockData for each colour in our palette. And we now have a phenomenal 31.25% more colours [gamut out of here - Ed] with which to play. To use this in the drawing loop, use the following code inside the **for** loops:

```
pixel = mcImage.getpixel((j,k))
if pixel < mcPaletteLength:
   bType = mcLookup[pixel][0]
   bData = mcLookup[pixel][1]
   mc.setBlock(j + x + 5, rheight - k + y, z, bType, bData)
```

In this manner you could add any blocks you like to your palette, but be careful with the lava and water ones: their pleasing orange and blue hues belie an inconvenient tendency to turn into lava/waterfalls. Incidentally, lava and water will combine to create obsidian. Cold, hard obsidian. ■

More dimensions

One of the earliest documentations of displaying custom images in *Minecraft:Pi Edition* is Dav Stott's excellent tutorial on displaying Ordnance Survey maps, **http://bit.ly/1lP2OE5**. Two-dimensional images are all very well, but Steve has a whole other axis to play with. To this end the aforementioned Ordnance Survey team has provided, for the full version of *Minecraft*, a world comprising most of Great Britain, with each block representing 50m. Its Danish counterpart has also done similar, though parts of *Minecraft*-Denmark were sabotaged by miscreants. Another fine example is Martin O'Hanlon's excellent 3d modelling project. This can import .obj files (text files with vertex, face and texture data) and display them in *Minecraft: Pi Edition*. Read all about it at **http://bit.ly/1sutoOS** . Of course, we also have a temporal dimension, so you could expand this tutorial in that direction, giving Steve some animated gifs to jump around on. If you were to proceed with this, then you'd probably have to make everything pretty small – the drawing process is slow and painful. Naturally, someone (Henry Garden) has already taken things way too far and has written *Redstone* – a Clojure interface to *Minecraft* which enables movies to be rendered. You can see the whole presentation including a blockified *Simpsons* title sequence at **http://bit.ly/1sO0A2q**.

Build 2048 in Minecraft

We show you how to implement a crude 2048 clone in Minecraft.

You may have not encountered the opium-like moreishness of the Android/iOS game *2048*, but once you do, you'll get a taste of the thrilling addition of successive powers of two and as an addict, like so many hopeless lovers, you'll be powerless to abandon the pursuit of the cherished 2048 tile, always just beyond reach. Granted, much of the pleasure comes from the touch interface: there is an innate pleasure in orchestrating an elegant and board-clearing sequence of cascades with just a few well-executed swipes. Be that as it may, the game has simple rules and is based on blocks. As such, it's perfect material for the next instalment in our *Minecraft:Pi Edition* series.

We'll worry about how to draw and move the blocks later. The first challenge is to understand the algorithm underlying the game. So dutifully read the rules in the box – they might seem clumsily worded or overly complex, but rest assured they have been carefully crafted for easy translation to Python. Sometimes it's easier to visualise the situation using pseudocode – this way we can see the shape of the program without getting bogged down in details.

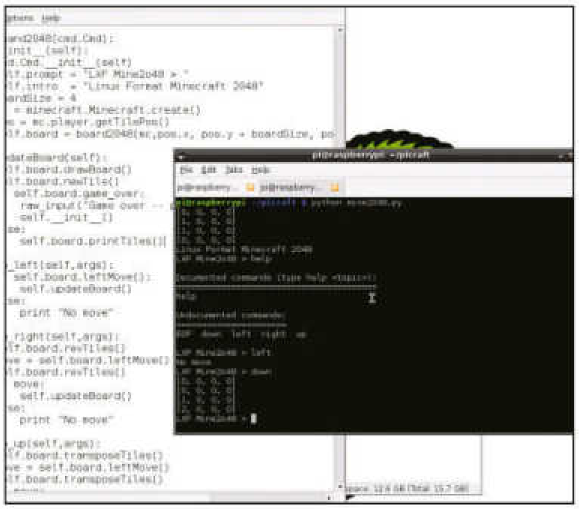
```
for each row:
    move tiles left, clearing empty space

for each row:
    for column 0 to 2:
        if tile[row,column + 1] = tile[row,column]:
            double tile[row, column]
            for x in column + 1 to 2:
                tile[row][x] = tile[row, x + 1]
            empty tile[row,3]

insert random 2 or 4 tile
if no move can be done:
    game over
```

Our task now is to flesh out the English bits of the above code, but to do this we need to state how we're going to represent the board and tiles. This will be done using a 4x4 array called **tiles**. The first co-ordinate will be the row number, and the second the column number, numbering the rows from top-to-bottom and the columns from left-to-right. Python uses 0-based lists, so **tiles[0][0]** will be the the top-left hand tile. Non-empty tiles will all be powers of 2, so we can use the exponent rather than the actual value here – for example, **32** will be represented by **5**. We can represent empty tiles using 0, so our array **tiles** is comprised entirely of (small) integers. Lovely.

Now we know what we're dealing with, we can set about doing some actual Python. We've gone for an object-oriented approach here, which means that there will be a lot of **self** flying around – but don't worry, it makes it easier for the *Minecraft* functions to talk to those to do with the board and

Quick tip

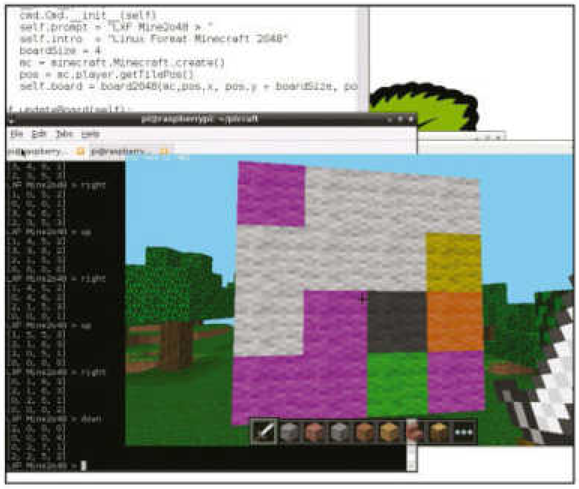
You could make this project a bit less of a *Zork*-esque text adventure by writing a control system based on player moves. You'd want to reset the player's position after each move, but in principle it's quite a simple modification.



```
ard2048(cmd.Cmd):
init__(self):
d.Cmd.__init__(self)
lf.prompt = "LXF Mine2048 > "
lf.intro  = "Linux Format Minecraft 2048"
ardSize = 4
 = minecraft.Minecraft.create()
s = mc.player.getTilePos()
lf.board = board2048(mc,pos.x, pos.y + boardSize, po

dateBoard(self):
lf.board.drawBoard()
lf.board.newTile()
 self.board.game_over:
  raw_input("Game over --
  self.__init__()
se:
  self.board.printTiles()

_left(self,args):
 self.board.leftMove():
  self.updateBoard()
se:
  print "No move"

_right(self,args):
lf.board.revTiles()
ve = self.board.leftMove()
lf.board.revTiles()
move:
  self.updateBoard()
se:
  print "No move"

_up(self,args):
lf.board.transposeTiles()
ve = self.board.leftMove()
lf.board.transposeTiles()
```

```
pi@raspberrypi ~/picraft $ python mine2048.py
[illegible]
Linux Format Minecraft 2048
LXF Mine2048 > help

Documented commands (type help <topic>):
========================================
help

Undocumented commands:
======================
EOF  down  left  right  up

LXF Mine2048 > left
No move
LXF Mine2048 > down
[illegible]
LXF Mine2048 >
```

❯ **You even get a handy help command, but it kind of looks like a cheat for *Sonic the Hedgehog*...**



```
cmd.Cmd.__init__(self)
self.prompt = "LXF Mine2048 > "
self.intro  = "Linux Format Minecraft 2048"
boardSize = 4
mc = minecraft.Minecraft.create()
pos = mc.player.getTilePos()
self.board = board2048(mc,pos.x, pos.y + boardSize, po
```

```
LXF Mine2048 > right
LXF Mine2048 > up
LXF Mine2048 > right
LXF Mine2048 > up
LXF Mine2048 > right
LXF Mine2048 > down
LXF Mine2048 >
```

The standard wool colour numbers don't really do high value blocks justice: eg 128 is represented by black.

the command interpreter. Honest. Let's look at line 2 of our pseudocode, in which we move tiles in the row left, occupying any available empty tiles. Python does have a **remove()** method for lists, but it only works on one element at a time, so we'll clear the zeroes from our row the old-fashioned way. We don't waste time fiddling with empty rows, and we also need to keep track of whether we actually moved anything (through the boolean **isMove**). Also don't worry about the three other possible moves for now; we can cater for them later with a very cunning trick.

```
def leftMove(self):
    isMove = False
    for row in range(self.boardSize):
        j = 0
        while j < self.boardSize - 1:
            # check rest of row non-empty
            row_empty = True
            for k in range(j,self.boardSize):
                if self.tiles[row][k] != 0:
                    row_empty = False
            if self.tiles[row][j] == 0 and not row_empty:
                for k in range(j,self.boardSize - 1):
                    self.tiles[row][k] = self.tiles[row][k + 1]
                self.tiles[row][self.boardSize - 1] = 0
                isMove = True
            else:
                j += 1
```

Now we can deal with the situation in the second block in the pseudocode, finding two horizontally-adjacent tiles are the same.

```
    for row in range(self.boardSize):
        for column in range(self.boardSize - 1):
            if self.tiles[row][column] == self.tiles[row][column + 1]
\
            and self.tiles[row][column] != 0:
                self.tiles[row][column] += 1
                for k in range(column + 1, self.boardSize - 1):
                    self.tiles[row][k]= self.tiles[row][k + 1]
                self.tiles[row][self.boardSize - 1] = 0
                isMove = True
    return isMove
```

Things turn out to be simpler if we split off the third pseudocode block into its own function, **newTile()**. We first need a list of co-ordinates of all the empty tiles; if there's more than one then it's certainly not Game Over, but even if there's only a single space (which will get filled by a new tile) there still may be possible moves. We use a couple of functions from the **random** module to help decide the where and what is of the new tile.

```
def newTile(self):
    empties = []
    self.game_over = True
    for j in range(self.boardSize):
        for k in range(self.boardSize):
            if self.tiles[j][k] == 0:
                empties.append([j,k])

    if len(empties) > 1:
        self.game_over = False

    rnd_pos = random.choice(empties)
    rnd_n = random.randint(1,2)
    self.tiles[rnd_pos[0]][rnd_pos[1]] = rnd_n
    self.drawBoard()
```

The **drawBoard()** function is what will actually place the blocks in *Minecraft* world. Checking whether moves are possible after the tile is added is a little awkward:

```
    # check row neighbours
    for j in range(self.boardSize):
        for k in range(self.boardSize - 1):
            if self.tiles[j][k] == self.tiles[j][k + 1]:
                self.game_over = False

    # check col neighbours
    for j in range(self.boardSize):
        for k in range(self.boardSize - 1):
            if self.tiles[k][j] == self.tiles[k + 1][j]:
                self.game_over = False
```

Quick tip

The board is updated twice for a successful move, before and after any tile merging. If you were feeling adventurous, you could use this to implement a two step animation by spacing out the tiles somehow. We were not feeling adventurous.

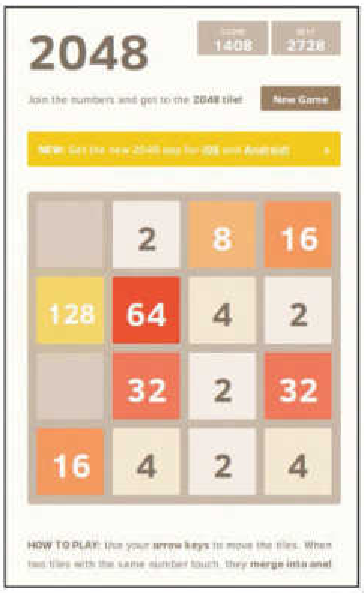
Get the code!

You can find the code at **http://www.linuxformat.com/files/mine2048.py.zip** or over at: **http://pastebin.com/5zEZUfBS**

2048: An Open Source Odyssey

The *2048* game was written by young Italian programmer Gabriele Cirulli, who wanted to see if he could write a game in a weekend. The fruits of his labour proved hugely popular, accruing over 4 million downloads within a week of its release in 2014. Numbers soared after mobile versions were released the following May, with up to 50,000 simultaneous players at its peak.

The game is described as a clone of Veewo Studio's *1024* and conceptually similar to the indie title *Threes!*. Rather than profit from his success, Cirulli's blog says he "didn't feel good about keeping [the code] private, since it was heavily based off someone else's work." Thus it is available to all at his GitHub **http://bit.ly/2048GitHub**. In the spirit of open source, this spurred all kinds of modifications and further depleted global productivity.



2048

SCORE 1408 BEST 2728

Join the numbers and get to the **2048 tile!**

New Game

NEW: Get the new 2048 app for iOS and Android!

	2	8	16
128	64	4	2
	32	2	32
16	4	2	4

HOW TO PLAY: Use your **arrow keys** to move the tiles. When two tiles with the same number touch, they **merge into one!**

It can be frustrating, and sometimes Steve's temper gets the better of him

Amazingly, that's the guts of the code all dealt with, but we still need a means by which to input our moves, and of course we need to render the results in *Minecraft* world.

Master and commander

We'll use the **cmd** module to provide a simple command line interface to our game. You may have seen this already in if you've ever made a *Minecraft* cannon and used **cmd** to control it (*You could always try making an image wall, p110*). We shall have a command for initialising the board, four directional commands and a command to quit, Ctrl-D, aka the end of file (EOF) character or the quick way to end a Python session. We even get a help command for free (see picture, page 114). When you enter a command, for example **left**, the cmd module will run the function with that name prefixed by **do_**, hence we have a function **do_left()** which calls the **leftMove()** function above. This function is really part of our **board** class.

The **do_left()** function will check if the left move is valid, and if so update the board, via the imaginatively-titled **updateBoard()** function, which in turn will add a new tile and decide whether or not the game is up. If it is all over, then we cheekily use **raw_input()** as a means to wait for the user to press Return. The cmd module works by subclassing its Cmd class. Any class you write that instantiates this will inherit all of its functionality. We use the standard boilerplate to start things up when the program is executed:

```
if __name__ == "__main__":
    command2048().cmdloop()
```

This will instantiate our command interpreter when the program is run with:

```
$ python mine2048.py
```

Minecraft must be running when you do this, or you'll run into errors. Also you'll need to make sure you've copied the Python API from your *Minecraft* install (e.g. **~/mcpi/api/python/mcpi**) to a subdirectory called **minecraft** in the same directory as the **mine2048.py** file from the download. But if you've been following our *Minecraft* series you'll be au fait with all this.

The **command2048** class's **__init__()** method sets up the basics, including setting up the **mc** object and using it to get the player's position. This provides a reference point for

Rules of the game

A key tenet of programming anything is knowing what the rules are – knowing exactly how your program should behave in a given set of circumstances. A vague idea won't cut it: we need specifics and exactitude, free of any semblance of ambiguity. With that in mind, let's consider how the game of *2048* works. We start with a 4x4 grid which has some 2s and some 4s on it at random locations. Initially two tiles are populated. The player can make one of four moves, shifting the tiles up, down, left or right. Suppose the player chooses left, then the following algorithm applies: For every row start by moving all the numbered tiles left so that any empty tiles are on the right of the row. Now we will look at each tile's situation and decide how the tiles will change. Starting with the tile on the left, if the tile to its right has the same value then the left-most of the pair will double in value, the rightmost of the pair will vanish, and all tiles to the right will move one position left. Repeat this process for the second and third tiles from the left. Repeat the whole process for the second, third and fourth rows. If there is still space then another 2 or 4 tile is added randomly in one of the available empty spaces. If no move is possible after this then it's Game Over.

For other directions, the same algorithm applies with strategic substitution of row with column and right with left, up or down.

drawing our board, which we will arbitrarily decide to be three blocks away in the z direction. Make sure you're not looking at the back of the board, though, as everything will be backwards. For convenience, we also have a **printTiles()** function which draws the board at the command terminal. We display only the exponents since (a) we're lazy and (b) it looks nicer if most things occupy just a single character. Change it if you like.

The **drawBoard** function uses different colours of wool (blockType 35) to represent the content of our array **tiles**:

```
def drawBoard(self):
   for j in range(self.boardSize):
      for k in range(self.boardSize):
         self.mc.setBlock(self.x + k, self.y - j, self.z, 35, self.
tiles[j][k])
```

This works reasonably, but feel free to add to it, for example making fancy glowing obsidian blocks for the higher valued ones. Note that our **row[0]** is the top one, so we count down, subtracting **j** from our y co-ordinate.

The only way is Left

But wait, an elephant in the room. We still haven't dealt with 75% of the moves allowed in *2048*. We could do this by copying the **leftMove()** function and painstakingly changing all the ranges and indices, and plus and minus signs, going through all the incorrect combinations until we find one that works. But that would be silly, and we don't tolerate such inefficiency here at Future Towers. Let us take a more grown-up approach. First, observe that moving the tiles to the right is exactly the same as reversing the rows, moving the tiles to the left, and then reversing the rows again. Reversing the rows is easy, once you figure out how to make a structural copy of our two-dimensional list:

```
def revTiles(self):
   oldTiles = [j[:] for j in self.tiles]
   for j in range(self.boardSize):
      for k in range(self.boardSize):
         self.tiles[j][k] = oldTiles[j][self.boardSize - k - 1]
```

So then we can make implement the (ahem) right move, in the following way:

```
def do_right(self,args):
   self.board.revTiles()
   move = self.board.leftMove()
   self.board.revTiles()
   if move:
      self.updateBoard()
   else:
      print "No move"
```

But the symmetry does not stop there. We can construct the Up move by transposing our tiles array (replacing each row by its respective column), then performing our **leftMove()** and transposing back again. Transposition is almost exactly as easy as row-reversal:

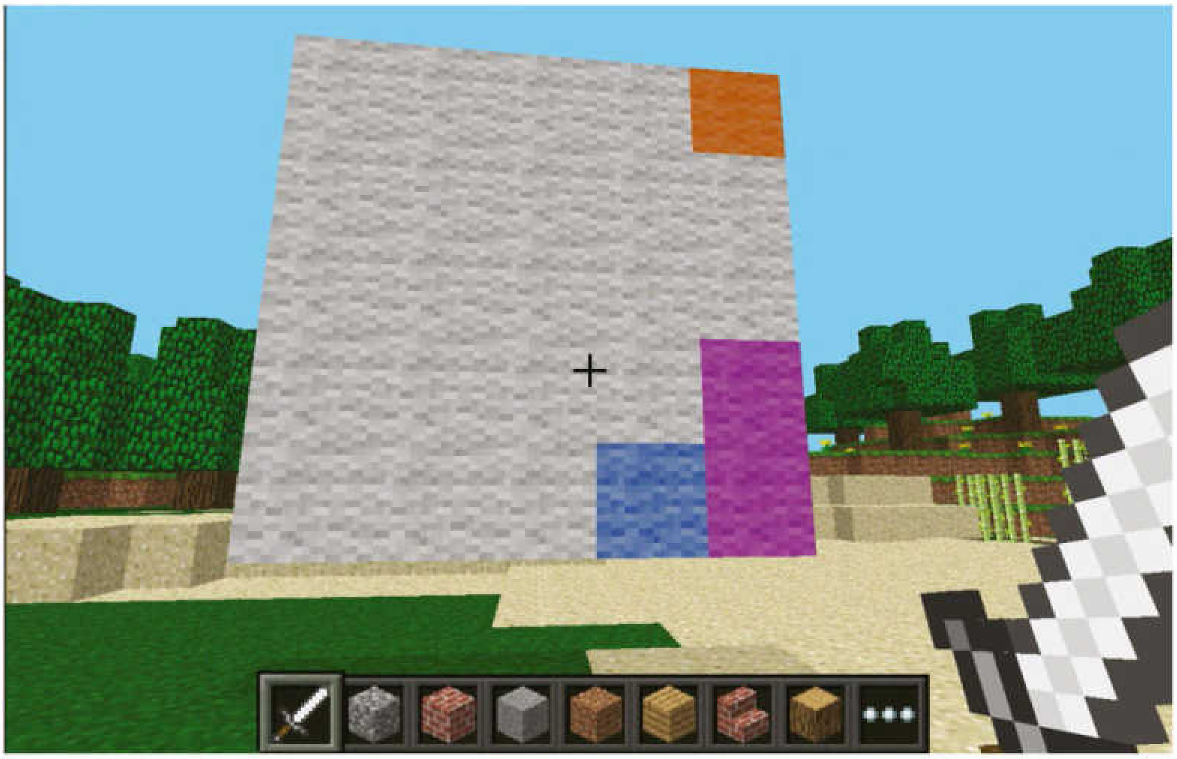
```
def transposeTiles(self):
   oldTiles = [j[:] for j in self.tiles]
   for j in range(self.boardSize):
      for k in range(self.boardSize):
         self.tiles[j][k] = oldTiles[k][j]
```

Similarly we can complete our directional command set, making the Down move by a combination of transposition, row reversal, left move, row reversal and transposition:

```
def do_down(self,args):
   self.board.transposeTiles()
   self.board.revTiles()
   move = self.board.leftMove()
   self.board.revTiles()
   self.board.transposeTiles()
```

Note that this is the same as transposition, right move and transposition.

Spotting wee tricks like this is great because they don't only make your program smaller (our provided code is about 160 lines long, not bad for a whole mini-game that can sap hours from your life). They also make it much easier to debug. The **leftMove()** function is by far the most complicated, so not having another three very similar functions is a considerable benefit. ■



› **You can change the boardSize variable, but it's a bit of a different game in a larger arena.**

CODING

MADE SIMPLE

Coding projects

Exciting and interesting coding projects for you to write

Python: Dive into version 3.0

Join us as we investigate what is probably one of the least loved and disregarded sequels in the whole history of programming languages.

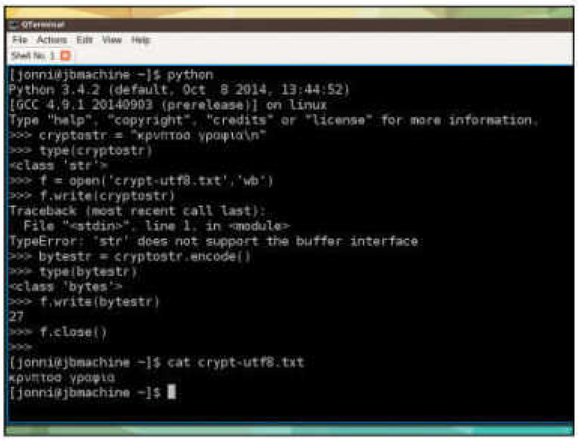
Way back in December 2008, Python 3.0 (also known as Py3k or Python 3000) was released. Yet here we are, seven years later, and most people are still not using it. For the most part, this isn't because Python programmers and distribution maintainers are a bunch of laggards, and the situation is very different from, for example, people's failure/refusal to upgrade (destroy?) Windows XP machines. For one thing, Python 2.7, while certainly the end of the 2.x line, is still regularly maintained, and probably will continue to be until 2020. Furthermore, because many of the major Python projects (also many, many minor ones) haven't been given the 3 treatment, anyone relying on them is forced to stick with 2.7.

Early on, a couple of big projects – NumPy and Django – did make the shift, and the hope was that other projects would follow suit, leading to an avalanche effect. Unfortunately, this didn't happen and most Python code you find out there will fail under Python 3. With a few exceptions, Python 2.7 is forwards-compatible with 3.x, so in many cases it's possible to come up with code that will work in both, but still programmers stick to the old ways. Indeed, even in *Linux Format,* certain authors, whether by habit, ignorance or affection for the past, continue to provide code that is entirely incompatible with Python 3. We won't do that in this article. We promise.

So let's start with what might have been your first ever Python program:

```
print 'Hello world'
```

Guess what – it doesn't work in Python 3 (didn't you just promise...?). The reason it doesn't work is that **print** in Python 2 was a statement, while in Python 3 **print** is a function, and functions are, without exception, called with brackets. Remember that functions don't need to return anything (those that don't are called void functions), so **print** is now a void function which, in its simplest form, takes a string as input, displays that string as text to *stdout*, and returns nothing. In a sense, you can pretend **print** is a function in Python 2, since you can call it with brackets, but a decision was made to offer its own special syntax and a bracketless shorthand. This is rather like the honour one receives in mathematics when something named after its creator is no longer capitalised – for example, abelian groups. But these kind of exceptions are not a part of the Python canon ("Special cases aren't special enough to break the rules"), so it's brackets all the way. On a deeper level, having a function-proper print does allow more flexibility for programmers – as a built-in function, it can be replaced, which might be useful if you're into defying convention or making some kind of Unicode-detecting/defying wrapper function. Your first Python program should have been:



```
[jonni@jbmachine ~]$ python
Python 3.4.2 (default, Oct  8 2014, 13:44:52)
[GCC 4.9.1 20140903 (prerelease)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> cryptostr = "κρυπτοσ γραφια\n"
>>> type(cryptostr)
<class 'str'>
>>> f = open('crypt-utf8.txt','wb')
>>> f.write(cryptostr)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' does not support the buffer interface
>>> bytestr = cryptostr.encode()
>>> type(bytestr)
<class 'bytes'>
>>> f.write(bytestr)
27
>>> f.close()
>>>
[jonni@jbmachine ~]$ cat crypt-utf8.txt
κρυπτοσ γραφια
[jonni@jbmachine ~]$ █
```

The Greek kryptos graphia, which translates as 'hidden writing', followed by a new line using the correct script.

The Unicode revolution

Traditionally, text was encoded in ASCII, in which each character is encoded as a 7-bit codepoint, which gives you 128 characters to play with. Some of these characters are invisible teletype codes (ASCII originated in the 1960s), and once we've counted the familiar alphanumeric characters, there isn't really much room left. Because we like things to be bytes, several 256-character extensions of the ASCII encoding emerged. The most notorious of these is ISO-8859-1, sometimes called Latin-1. This widely-used character set (and the related Windows-1252) contains almost all the accents required for the Latin-scripted languages, as well as the characters used in the romanisation of other languages. As a result, it's fairly common in the western hemisphere, but doesn't really solve the problem elsewhere.

The correct solution would be a standard encoding (or maybe a couple of them) that accounts for as many as possible of the set of characters anyone on earth might conceivably wish to type. Obviously, this will require many more than 256 characters, so we'll have to do away with one character encoding to one byte (hence the divergence of codepoints and byte encodings), but it's for a greater good. Fortunately, all the wrangling, tabulating and other rigmarole has been done, and we have an answer: Unicode. This accounts for over 100,000 characters, bidirectional display order, ligature forms and more. Currently there are two encodings in use: UTF-8, which uses one byte for common characters (making it entirely backwards compatible with ASCII), and up to four bytes for the more cosmopolitan ones; and UTF-16, which uses two bytes for some characters and four bytes for others. Unicode has been widely adopted, both as a storage encoding standard and for internally processing tests. The main raison d'être of Python 3 is that its predecessor did not do the latter.

```
print ('Hello world')
```

which is perfectly compatible with Python 2 and 3. If you were a fan of using a comma at the end of your print statements (to suppress the newline character), then sad news: this no longer works. Instead, we use the **end** parameter, which by default is a new line. For example:

```
print ('All on', end=" ")
print ('one line')
```

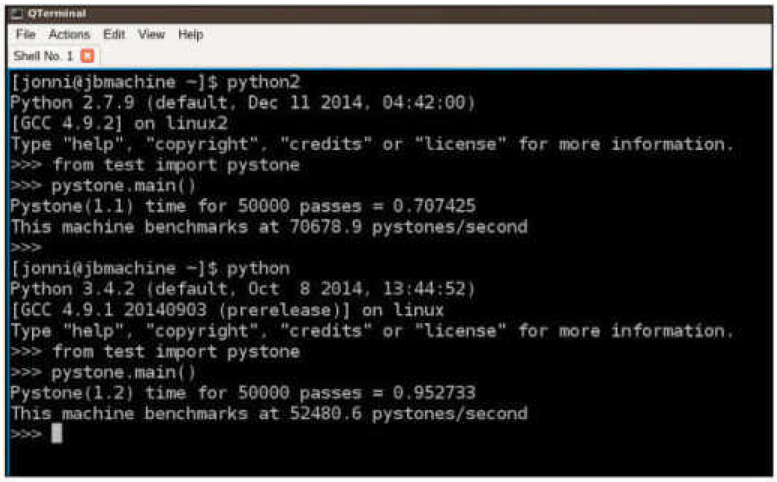
does just that.

Print in Python 3

A significant proportion of Python programs could be made compatible with 3 just by changing the **print** syntax, but there are many other, far less trivial, things that could go wrong. To understand them, we must first be au fait with what really changed in Python 3.

Most of the world doesn't speak English. In fact, most of the world doesn't even use a Latin character set; even those regions that do tend to use different sets of accents to decorate the characters. As a result, besides the ASCII standard, numerous diverse and incompatible character encodings have emerged. Each grapheme (an abstraction of a character) is assigned a codepoint, and each codepoint is assigned a byte encoding, sometimes identically. In the past, if you wanted to share a document with foreign characters in it, then plain ASCII wouldn't help. You could use one of the alternative encodings, if you knew the people you were sharing it with could do the same, but in general you needed to turn to a word processor with a particular font, which just moves the problem elsewhere. Thankfully, we now have a widely adopted standard: Unicode (*see The Unicode Revolution box, above*) that covers all the bases, and is backwards compatible with ASCII and (as far as codepoints are concerned) its Latin-1 extension. We can even have Unicode in our domain names, although internally these are all still encoded as ASCII, via a system called Punycode.

Python 2 is far from devoid of Unicode support, but its handling of it is done fairly superficially (Unicode strings are sneakily re-encoded behind the scenes) and some third-party modules still won't play nicely with it. Strings in Python 2 can be of type **str** (which handles ASCII fine, but will behave unpredictably for codepoints above 127) or they can be of type **unicode**. Strings of type **str** are stored as bytes and, when printed to a terminal, are converted to whichever encoding your system's locale specified (through the **LANG** and **LC_*** environment variables in Linux). For any modern distro, this is probably UTF-8, but it's definitely not something you should take for granted.



```
[jonni@jbmachine ~]$ python2
Python 2.7.9 (default, Dec 11 2014, 04:42:00)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from test import pystone
>>> pystone.main()
Pystone(1.1) time for 50000 passes = 0.707425
This machine benchmarks at 70678.9 pystones/second
>>>
[jonni@jbmachine ~]$ python
Python 3.4.2 (default, Oct  8 2014, 13:44:52)
[GCC 4.9.1 20140903 (prerelease)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from test import pystone
>>> pystone.main()
Pystone(1.2) time for 50000 passes = 0.952733
This machine benchmarks at 52480.6 pystones/second
>>> 
```

The *PyStone* benchmark will likely be slower in Python 3, but the same won't be true for all code. Don't be a Py3k refusenik without first trying your code.

The **unicode** type should be used for textual intercourse – finding the length of, slicing or reversing a string. For example, the Unicode codepoint for the lowercase Greek letter pi is **03c0** in hex notation. So we can define a unicode string from the Python console like so, provided our terminal can handle Unicode output and is using a suitable font:

```
>>> pi = u'\u03c0'
>>> print(pi)
ϖ
>>> type(pi)
<type 'unicode'>
>>> len(pi)
1
```

However, if we were to try this on a terminal without Unicode support, things will go wrong. You can simulate such a scenario by starting Python with:

```
$ LC_ALL=C python
```

Now when you try to print the lowercase character pi, you will »

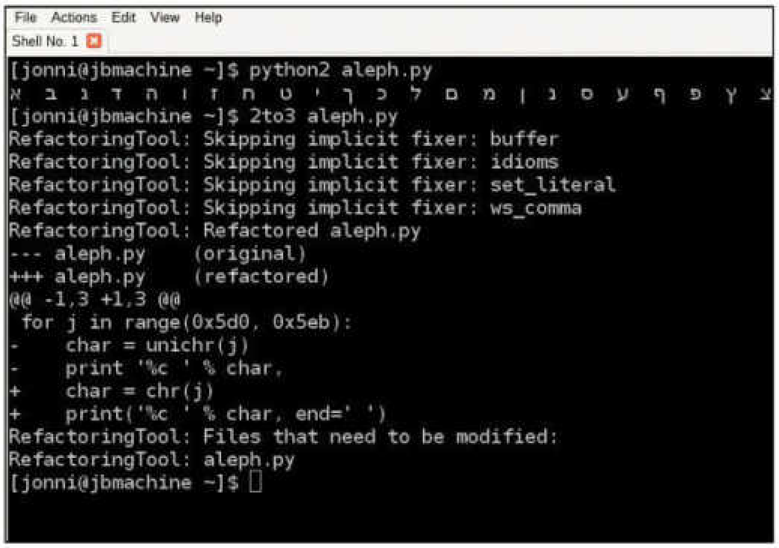
Quick tip

Arch Linux is one of few distributions to use Python 3 by default, but it can live happily in tandem with its predecessor (available in the **python2** package).

» run into a **UnicodeEncodeError**. Essentially, Python is trying and failing to coerce this to an ASCII character (the only type supported by the primitive C locale). Python 2 also tries to perform this coercion (regardless of current locale settings) when printing to a file or a pipe, so don't use the **unicode** type for these operations, instead use **str**.

The **str** type in Python 2 is really just a list of bytes corresponding to how the string is encoded on the machine. This is what you should use if you're writing your strings to disk or sending them over a network or to a pipe. Python 2 will try and convert strings of type **unicode** to ASCII (its default encoding) in these situations, which could result in tears. So we can also get a funky pi character by using its UTF-8 byte representation directly. There are rules for converting Unicode codepoints to UTF-8 (or UTF-16) bytes, but it will suffice to simply accept that the pi character encodes to the two bytes **CF 80** in UTF-8. We can escape these with an **\x** notation in order to make Python understand bytes:

```
>>> strpi = '\xCF\x80'
>>> type(strpi)
<type 'str'>
>>> len(strpi)
2
```



```
[jonni@jbmachine ~]$ python2 aleph.py
א  ב  ג  ד  ה  ו  ז  ח  ט  י  ך  כ  ל  ם  מ  ן  נ  ס  ע  ף  פ  ץ  צ
[jonni@jbmachine ~]$ 2to3 aleph.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
RefactoringTool: Refactored aleph.py
--- aleph.py    (original)
+++ aleph.py    (refactored)
@@ -1,3 +1,3 @@
 for j in range(0x5d0, 0x5eb):
-    char = unichr(j)
-    print '%c ' % char,
+    char = chr(j)
+    print('%c ' % char, end=' ')
RefactoringTool: Files that need to be modified:
RefactoringTool: aleph.py
[jonni@jbmachine ~]$ 
```

› **Aleph, beth, gimel... The *2to3* program shows us how to refactor our Python 2 code for Python 3. Shalom aleikhem.**

So ϖ apparently now has two letters. The point is: if your Python 2 code is doing stuff with Unicode characters, you'll need to have all kinds of wrappers and checks in place to take account of the localisation of whatever machine may run it. You'll also have to handle your own conversions between **str** and **unicode** types, and use the codecs module to change encodings as required. Also, if you have Unicode strings in your code, you'll need to add the appropriate declaration at the top of your code:

```
# -*- coding: utf-8 -*-
```

The main driving force for a new Python version was the need to rewrite from the ground up how the language dealt with strings and their representations to simplify this process. Some argue that it fails miserably (see Armin Ronacher's rant at **http://bit.ly/UnicodeInPython3**), but it really depends on your purposes. Python 3 does away with the old unicode type entirely, since everything about Python 3 is now Unicode-orientated. The str type now stores Unicode codepoints, the language's default encoding is UTF-8 (so no need for the **-*- coding** decorator above) and the new **bytes** object stores byte arrays, like the old str type. The new str type will need to be converted to bytes if it's to be used for any kind of file I/O, but this is trivial via the **str.encode()** method. If you're reading Unicode text files, you'll need to open them in binary 'rb' mode and convert the bytes to a string using the converse **bytes.decode()** method (*see the picture for details, bottom p120*).

Python 3, however, brings about many other changes besides all this Unicode malarkey, some of these are just the removal of legacy compatibility code (Python 3, unlike 2.7, doesn't need to be compatible with 2.0), some of them provide new features, and some of them force programmers to do things differently.

But wait, there's more: for example, functions can now be passed keyword-only arguments, even if they use the ***args** syntax to take variable length argument lists, and catching exceptions via a variable now requires the **as** keyword. Also, you can no longer use the ugly **<>** comparison operator to test for inequality – instead use the much more stylish **!=** which is also available in Python 2.

Automating conversion with 2to3

For reasonably small projects, and those that aren't going to encounter any foreign characters, it's likely that your Python 2 code can be automatically made Python 3 compatible using the *2to3* tool. This helpful chap runs a predefined set of fixers

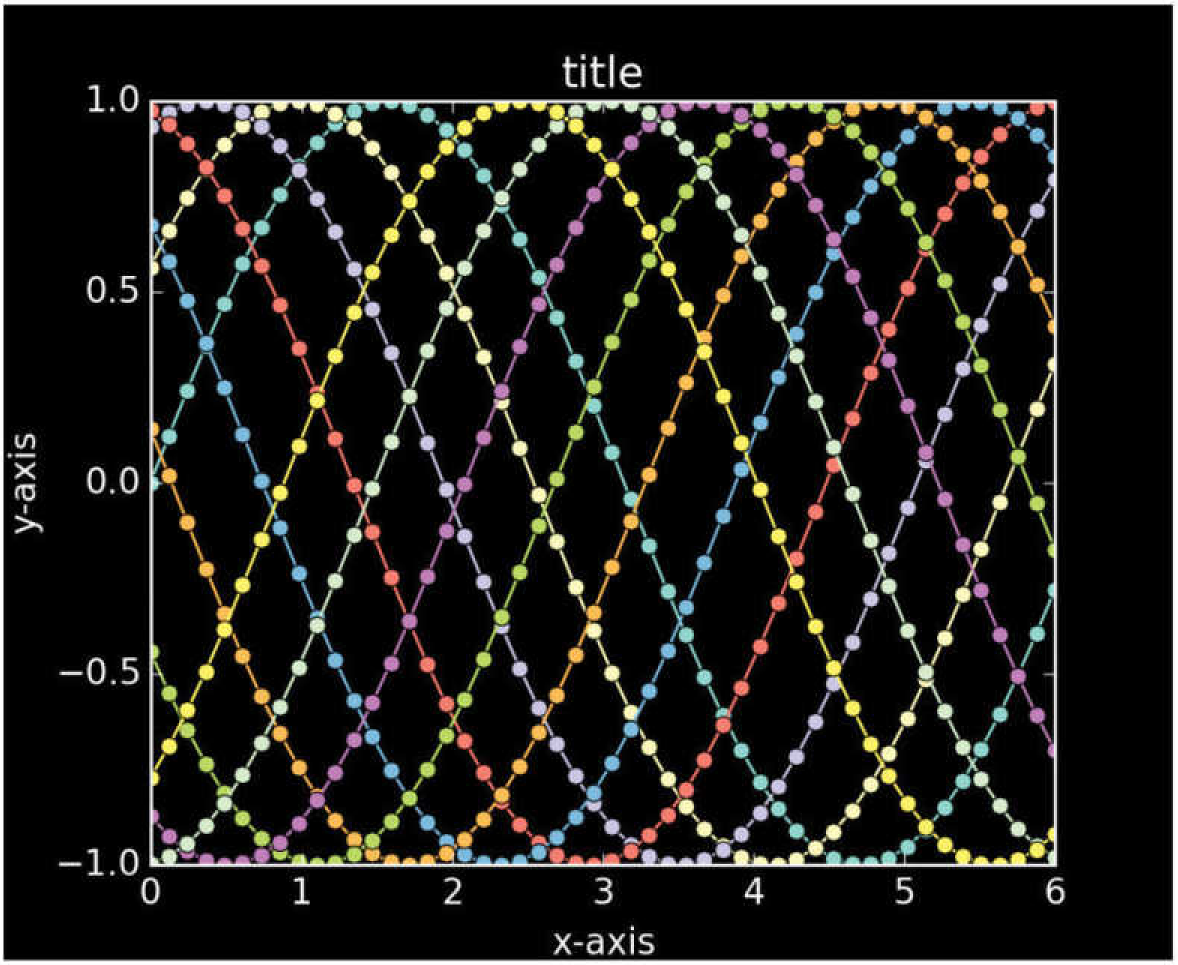
The division bell

One change in Python 3 that has the potential to flummox is that the behaviour of the division operator **/** is different. For example, here's an excerpt from a Python 2 session

```
>>> 3/2
1
>>> 3./2
1.5
>>> 3.//2
1.0
```

which shows the first command operating on two ints, and returning an int, thus in this case the operator stands for integer division. In the second example, the numerator is a float and the operator acts as floating point division, returning what you'd intuitively expect half of three to be. The third line uses the explicit floor division operator **//** which will return the rounded-down result as a float or an int, depending on the arguments. The latter operator was actually backported to 2.7, so its behaviour is the same in Python 3, but the behaviour of the classic division operator has changed: if both numerator and denominator are ints, then an int is returned if one divides the other, otherwise a float is returned with the correct decimal remainder. If at least one of the arguments is a float (or complex), then the behaviour is unchanged.

This means the **/** operator is now as close to mathematical division proper as we can hope, and issues involving unexpected ints will no longer cause harm.



The popular matplotlib module has been Python 3 compatible since v1.2, for all your graphing and plotting requirements.

on your Python 2 code, with the goal of emitting bona fide Python 3 code. Using it can be as simple as

```
$ 2to3 mypy2code.py
```

which will output a diff of any changes against the original file. You can also use the **-w** option to overwrite your file – don't worry, it will be backed up first. Some fixers will only run if you specify them explicitly, using the **-f** option. An example is **buffer** which will replace all **buffer** types with **memoryviews**. These two aren't entirely compatible, so some tweaking might be necessary in order to successfully migrate. Using *2to3* can save you a whole heap of effort, because searching and replacing **print** statements manually, for instance, is a drudgerous task. The pictured example (*over on the left, on p122*) shows the changes to a simple three-line program – the **unichr()** function is now **chr()**, because Unicode is now implicit, and the print line is reworked, even if it uses **%** to format placeholders.

Parlez-vous Python Trois?

Another option is to create 'bilingual' code that is compatible with both Python 2 and Python 3. While you should really only target one specific Python version, this middle ground is very useful when you're porting and testing your code. You might need to rework a few things in Python 2 while still enjoying the new features and learning the new syntax. Many Python 3 features and syntaxes have already been backported to 2.7, and many more can be enabled using the __future__ module. For example, you can get the new-fangled **print** syntax by using the following:

```
>>> from __future__ import print_function
```

This is used with the **from** in this way, and __future__ doesn't behave like a standard module – instead it acts as a compiler directive, which in this case provides the modern print syntax. We can likewise import division to get the new style division, or unicode_literals to make strings Unicode by default. There is another module, confusingly called future, which isn't part of the standard library, but can help ease transition issues.

When Python 3.0 was released, it was generally regarded as being about 10% slower than its predecessor. This was not surprising, because speed was not really a priority for the new version and many special-case optimisations were removed in order to clean up the code base. Now that we're up to version 3.5 (which was released in September 2015), it would be nice if performance had since been improved. Unfortunately, this hasn't been the case, which you can verify for yourself using the *PyStone* benchmark.

We tested it (*see the screenshot on p121)* on an aging machine in the office, which has now come back from the dead twice and hence possesses supernatural powers far beyond its dusty 2.3GHz processor would suggest. But don't be misled; *PyStone* tests various Python internals, of which your code may or may not make extensive use. It's important to test your code in both versions to get a more accurate picture. You can always use Cython (maintained at **http://cython.org**) to speed up code that is amenable to C translation (loops, math, array access), or the bottleneck module.

Guido van Rossum, the author of Python, says that Python 2.7 will be supported until 2020, but that's no excuse to postpone learning the new version now. Python 3 adoption is increasing, so you won't be alone. Half the work is retraining your fingers to add parentheses. ■

Python: Code a Gimp plugin

Use Python to add some extra features to the favourite open source image-manipulation app, without even a word about *Gimp* masks.

Multitude of innuendoes aside, *Gimp* enables you to extend its functionality by writing your own plugins. If you wanted to be hardcore, then you would write the plugins in C using the *libgimp* libraries, but that can be pretty off-putting or rage-inducing. Mercifully, there exist softcore APIs to *libgimp* so you can instead code the plugin of your dreams in *Gimp*'s own Script-Fu language (based on Scheme), Tcl, Perl or Python. This tutorial will deal with the last in this list, which is probably most accessible of all these languages, so even if you have no prior coding experience you should still get something out of it.

Get started

On Linux, most packages will ensure that all the required Python gubbins get installed alongside *Gimp*; your Windows and Mac friends will have these included as standard since version 2.8. You can check everything is ready by starting up *Gimp* and checking for the Python-Fu entry in the Filters menu. If it's not there, you'll need to check your installation. If it is there, then go ahead and click on it. If all goes to plan this should open up an expectant-looking console window, with a prompt (>>>) hungry for your input. Everything that *Gimp* can do is registered in something called the Procedure Database (PDB). The Browse button in the console window will let you see all of these procedures, what they operate on and what they spit out. We can access every single one of them in Python through the **pdb** object.

As a gentle introduction, let's see how to make a simple blank image with the currently selected background colour. This involves setting up an image, adding a layer to it, and then displaying the image.

```
image = pdb.gimp_image_new(320,200,RGB)
layer = pdb.gimp_layer_new(image,320,200,RGB,'Layer0',100
,RGB_IMAGE)
pdb.gimp_image_insert_layer(image,layer,None,0)
pdb.gimp_display_new(image)
```

So we have used four procedures: **gimp_image_new()**, which requires parameters specifying width, height and image type (RGB, GRAY or INDEXED); **gimp_layer_new()**, which works on a previously defined image and requires width, height and type data, as well as a name, opacity and combine mode; **gimp_image_insert_layer()** to actually add the layer to the image, and **gimp_display_new()**, which will display an image. You need to add layers to your image before you can do anything of note, since an image without layers is a pretty ineffable object. You can look up more information about these procedures in the Procedure Browser – try typing **gimp-layer-new** into the search box, and you will see all the different combine modes available. Note that in Python, the hyphens in procedure names are replaced by underscores, since hyphens are reserved for subtraction. The search box will still understand you if you use underscores there, though.

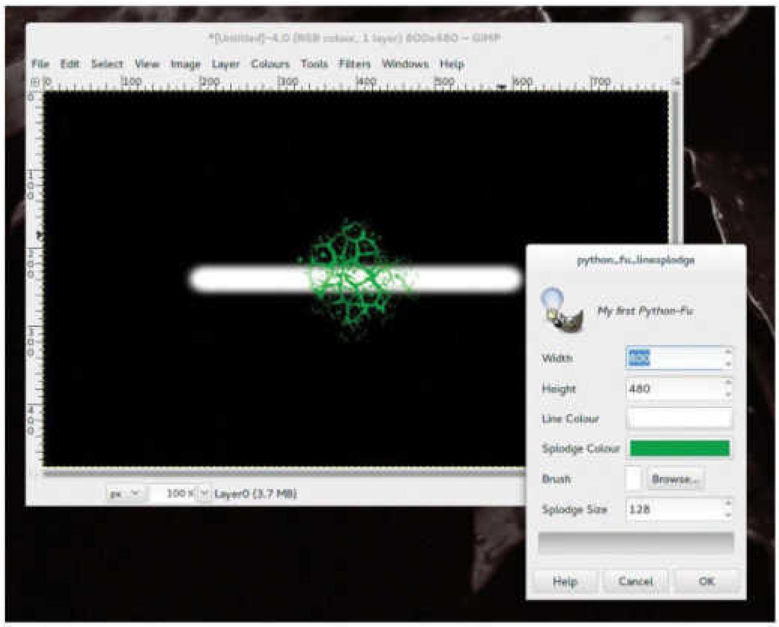
Draw the line

All well and good, but how do we actually draw something? Let's start with a simple line. First select a brush and a foreground colour that will look nice on your background. Then throw the following at the console:

```
pdb.gimp_pencil(layer,4,[80,100,240,100])
```

Great, a nicely centred line, just like you could draw with the pencil tool. The first parameter, **gimp_pencil()**, takes just the layer you want to draw on. The syntax specifying the points is a little strange: first we specify the number of coordinates, which is twice the number of points because each point has an x and a y component; then we provide a list of the form [x1, y1, ..., xn, yn]. Hence our example draws a line from (80,100) to (240,100). The procedures for selecting and adjusting colours, brushes and so forth are in the PDB too:

```
pdb.gimp_context_set_brush('Cell 01')
```


*[Untitled]-4.0 (RGB colour, 1 layer) 800x480 – GIMP
File Edit Select View Image Layer Colours Tools Filters Windows Help
px
100 %
Layer0 (3.7 MB)
python_fu_linesplodge
My first Python-Fu
Browse...
Help
Cancel
OK

Width [illegible]
Height 480
Line Colour
Splodge Colour 🟩
Brush
Splodge Size 128

› **You can customise lines and splodges to your heart's content, though frankly doing this is unlikely to produce anything particularly useful.**

› **Applying our bokeh plugin has created a pleasing bokeh effect in the highlights.**

```
pdb.gimp_context_set_foreground('#00a000')
pdb.gimp_context_set_brush_size(128)
pdb.gimp_paintbrush_default(layer,2,[160,100])
```

If you have the brush called 'Cell 01' available, then the above code will draw a green splodge in the middle of your canvas. If you don't, then you'll get an error message. You can get a list of all the brushes available to you by calling **pdb.gimp_brushes_get_list('')**. The paintbrush tool is more suited to these fancy brushes than the hard-edged pencil, and if you look in the procedure browser at the function **gimp_paintbrush**, you will see that you can configure gradients and fades too. For simplicity, we have just used the defaults/current settings here. Download the code pack from **http://bit.ly/TMS12code** and you will find a file *linesplodge.py* which will register this a fully-fledged *Gimp* plugin, replete with a few tweaks.

For the rest of the tutorial we will describe a slightly more advanced plugin for creating bokeh effects in your own pictures. 'Bokeh' derives from a Japanese word meaning blur or haze, and in photography refers to the out-of-focus effects caused by light sources outside of the depth of field. It often results in uniformly coloured, blurred, disc-shaped artefacts in the highlights of the image, reminiscent of lens flare. The effect you get in each case is a characteristic of the lens and the aperture – depending on design, one may also see polygonal and doughnut-shaped bokeh effects. For this exercise, we'll stick with just circular ones.

Our plugin will have the user pinpoint light sources using a path on their image, which we will assume to be single-layered. They will specify disc diameter, blur radius, and hue and saturation adjustments. The result will be two new layers: a transparent top layer containing the 'bokeh discs', and a layer with a blurred copy of the original image. The original layer remains untouched beneath these two. By adjusting the opacities of these two new layers, a more pleasing result may be achieved. For more realistic bokeh effects, a part of the image should remain in focus and be free of discs, so it may »

> **"Our plugin creates a layer with 'bokeh discs' and another with a blurred copy of the image"**

be fruitful to erase parts of the blurred layer. Provided the user doesn't rename layers, then further applications of our plugin will not burden them with further layers. This means that one can apply the function many times with different parameters and still have all the flare-effect discs on the same layer. It is recommended to turn the blur parameter to zero after the first iteration, since otherwise the user would just be blurring the already blurred layer.

After initialising a few de rigueur variables, we set about making our two new layers. For our blur layer, we copy our original image and add a transparency channel. The bokeh layer is created much as in the previous example.

```
    blur_layer = pdb.gimp_layer_copy(timg.layers[0],1)
    pdb.gimp_image_insert_layer(timg, blur_layer, None, 0)
    bokeh_layer = pdb.gimp_layer_new(timg, width, height,
 RGBA_IMAGE, "bokeh", 100, NORMAL_MODE)
    pdb.gimp_image_insert_layer(timg, bokeh_layer, None, 0)
```

Our script's next task of note is to extract a list of points from the user's chosen path. This is slightly non-trivial since a general path could be quite a complicated object, with curves and changes of direction and allsorts. Details are in the box below, but don't worry – all you need to understand is that the main **for** loop will proceed along the path in the order drawn, extracting the coordinates of each component point as two variables x and y.

Having extracted the point information, our next challenge is to get the local colour of the image there. The PDB function for doing just that is called **gimp_image_pick_color()**. It has a number of options, mirroring the dialog for the Colour Picker tool. Our particular call has the program sample within a 10-pixel radius of the point x,y and select the average colour. This is preferable to just selecting the colour at that single pixel, since it may not be indicative of its surroundings.

Bring a bucket

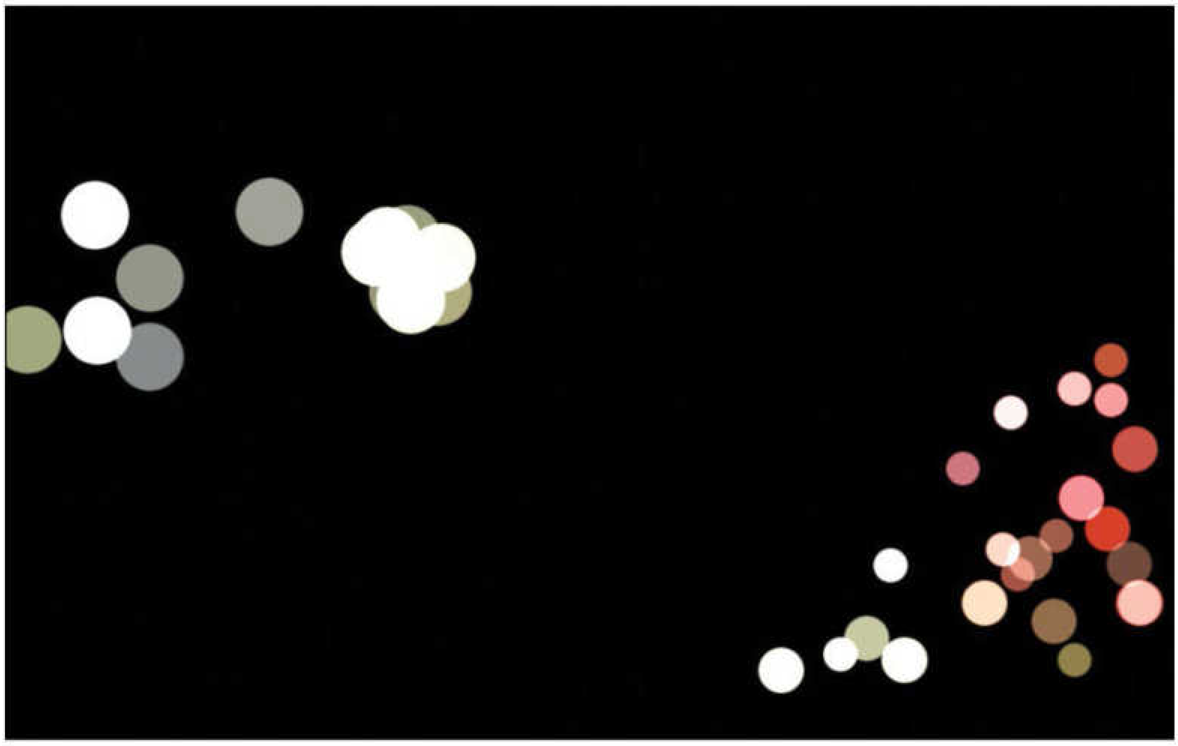
To draw our appropriately-coloured disc on the bokeh layer, we start – somewhat counter-intuitively – by drawing a black disc. Rather than use the paintbrush tool, which would rely on all possible users having consistent brush sets, we will make our circle by bucket filling a circular selection. The selection is achieved like so:

```
    pdb.gimp_image_select_ellipse(timg, CHANNEL_OP_
 REPLACE, x - radius, y - radius, diameter, diameter)
```

There are a few constants that refer to various *Gimp*-specific modes and other arcana. They are easily identified by their shouty case. Here the second argument stands for the

Quick tip

For many, many more home-brewed plugins, check out the *Gimp* Plugin Registry at **http://registry.gimp.org**



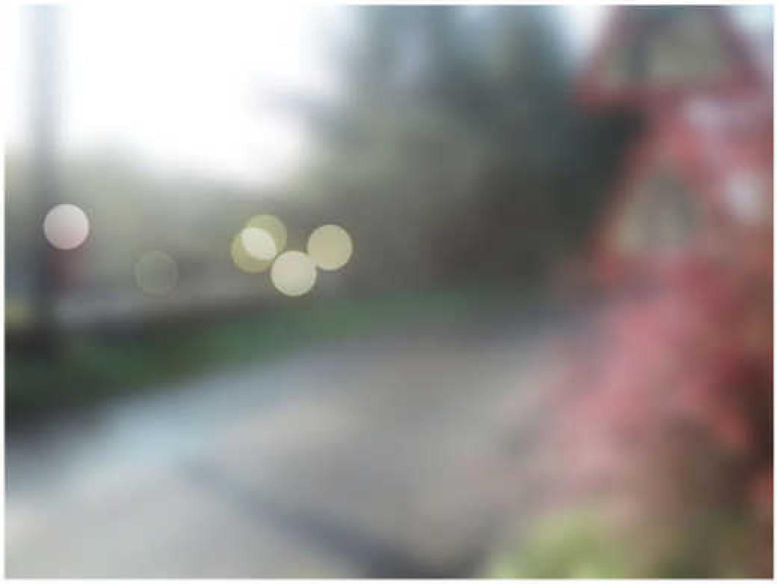
Here are our discs. If you're feeling crazy you could add blends or gradients, but uniform colour works just fine.

Paths, vectors, strokes, points, images and drawables

Paths are stored in an object called **vectors**. More specifically, the object contains a series of strokes, each describing a section of the path. We'll assume a simple path without any curves, so there is only a single stroke from which to wrest our coveted points. In the code we refer to this stroke as **gpoints**, which is really a tuple that has a list of points as its third entry. Since Python lists start at 0, the list of points is accessed as **gpoints[2]**. This list takes the form [x0,y0,x0,y0,x1,y1,x1,y1,...]. Each point is counted twice, because in other settings the list needs to hold curvature information. To avoid repetition, we use the **range()** function's step parameter to increment by 4 on each iteration, so that we get the xs in positions 0, 4, 8 and the ys in positions 1, 5, 9. The length of the list of points is bequeathed to us in the second entry of **gpoints**

```
for j in range(0,gpoints[1],4):
```

You will see a number of references to variables **timg** and **tdraw**. These represent the active image and layer (more correctly image and drawable) at the time our function was called. As you can imagine, they are quite handy things to have around because so many tools require at least an image and a layer to work on. So handy, in fact, that when we come to register our script in *Gimp*, we don't need to mention them – it is assumed that you want to pass them to your function. Layers and channels make up the class called drawables – the abstraction is warranted here since there is much that can be applied equally well to both.



After we apply the filter, things get a bit blurry. Changing the opacity of the layer will bring back some detail.

number 2, but also to the fact that the current selection should be replaced by the specified elliptical one.

The dimensions are specified by giving the top left corner of the box that encloses the ellipse and the said box's width. We feather this selection by two pixels, just to take the edge off, and then set the foreground colour to black. Then we bucket fill this new selection in Behind mode so as not to interfere with any other discs on the layer:

```
pdb.gimp_selection_feather(timg, 2)
pdb.gimp_context_set_foreground('#000000')
pdb.gimp_edit_bucket_fill_full(bokeh_layer, 0,BEHIND_
MODE,100,0,False,True,0,0,0)
```

And now the reason for using black: we are going to draw the discs in additive colour mode. This means that regions of overlapping discs will get brighter, in a manner which vaguely resembles what goes on in photography. The trouble is, additive colour doesn't really do anything on transparency, so we black it up first, and then all the black is undone by our new additive disc.

```
pdb.gimp_context_set_foreground(color)
pdb.gimp_edit_bucket_fill_full(bokeh_layer, 0,ADDITION_
MODE,100,0,False,True,0,0,0)
```

Once we've drawn all our discs in this way, we do a Gaussian blur – if requested – on our copied layer. We said that part of the image should stay in focus; you may want to work on this layer later so that it is less opaque at regions of interest. We deselect everything before we do the fill, since otherwise we would just blur our most-recently drawn disc.

```
if blur > 0:
    pdb.plug_in_gauss_iir2(timg, blur_layer, blur, blur)
```

Softly, softly

Finally we apply our hue and lightness adjustments, and set the bokeh layer to Soft-Light mode, so that lower layers are illuminated beneath the discs. And just in case any black survived the bucket fill, we use the Color-To-Alpha plugin to squash it out.

```
pdb.gimp_hue_saturation(bokeh_layer, 0, 0, lightness,
saturation)
pdb.gimp_layer_set_mode(bokeh_layer, SOFTLIGHT_
MODE)
pdb.plug_in_colortoalpha(timg, bokeh_layer, '#000000')
```

And that just about summarises the guts of our script. You will see from the code on the disc that there is a little bit of housekeeping to take care of, namely grouping the whole series of operations into a single undoable one, and restoring any tool settings that are changed by the script. It is always good to tidy up after yourself and leave things as you found them. In the **register()** function, we set its menupath to '<Image>/Filters/My Filters/PyBokeh...' so that if it registers correctly you will have a My Filters menu in the Filters menu. You could add any further scripts you come up with to this menu to save yourself from cluttering up the already crowded Filters menu. The example images show the results of a couple of PyBokeh applications.

> **"To finish, group the operations into a single undoable one, and reset any changed tool settings"**

Critics may proffer otiose jibes about the usefulness of this script, and indeed it would be entirely possible to do everything it does by hand, possibly even in a better way. That is, on some level at least, true for any *Gimp* script. But this manual operation would be extremely laborious and error-prone – you'd have to keep a note of the coordinates and colour of the centre of each disc, and you'd have to be incredibly deft with your circle superpositioning if you wanted to preserve the colour addition. ■

Registering your plugin

In order to have your plugin appear in the *Gimp* menus, it is necessary to define it as a Python function and then use the **register()** function. The tidiest way to do this is to save all the code in an appropriately laid out Python script. The general form of such a thing is:

```
#! /usr/bin/env python
from gimpfu import *
def myplugin(params):
    # code goes here
register(
    proc_name, # e.g. "python_fu_linesplodge"
    blurb, #."Draws a line and a splodge"
    help, author, copyright, date,
    menupath, imagetypes,
    params, results,
    function) # "myplugin"
main()
```

The **proc_name** parameter specifies what your plugin will be called in the PDB; 'python_fu' is actually automatically prepended so that all Python plugins have their own branch in the taxonomy. The **menupath** parameter specifies what kind of plugin you're registering, and where your plugin will appear in the *Gimp* menu: in our case "<Image>/Filters/Artistic/LineSplodge..." would suffice. **imagetypes** specifies what kind of images the plugin works on, such as "RGB*, GRAY*", or simply "" if it doesn't operate on any image, such as in our example. The list **params** specifies the inputs to your plugin: you can use special Python-Fu types here such as **PF_COLOR** and **PF_SPINNER** to get nice interfaces in which to input them. The **results** list describes what your plugin outputs, if anything. In our case **(PF_IMAGE, image, "LSImage")** would suffice. Finally, **function** is just the Python name of our function as it appears in the code.

To ensure that *Gimp* finds and registers your plugin next time it's loaded, save this file as (say) **myplugin.py** in the plugins folder: **~/.gimp-2.8/plug-ins** for Linux (ensure it is executable with **chmod +x myplugin.py**) or **%USERPROFILE%\.gimp-2.8\plug-ins\** for Windows users, replacing the version number as appropriate.

Pillow: Edit images easily

You can save time and effort by using Python to tweak your images in bulk.

Python's multimedia prowess extends to images and includes mechanisms for manipulating snapshots in a variety of ways. While it can be done manually, using Python enables you to automate the process by easily writing scripts to act on a disk full of images and carry out extensive modifications based on a complex set of rules.

The Python Image Library, commonly referred to as PIL, was a popular and powerful Python library for handling images. But it hasn't been updated for over half a decade, and doesn't support Python 3. Pillow is a fork of PIL that's based on the PIL code, and has evolved to be a better, modern and more friendly version of PIL. With Pillow, you can open, manipulate and save in all the popular image file formats. One of the best advantages of Pillow for existing PIL users is the similarity of the API between the two. Many of the existing PIL code that you'll find on the internet will work with Pillow with minimal or even no modifications.

To get started, you need to install the library with `sudo pip install Pillow`. The library has a bunch of modules. Many of the commonly used functions are found in the Image module. You can create instances of this class in several ways, either by loading images from files, processing other images, or creating images from scratch. As with all Python libraries, begin your code by importing the Pillow modules you want to use, such as:

```
from PIL import Image
```

You can now open an image with the `Image.open` method. Launch the Python interpreter and type:

```
>>> from PIL import Image
>>> im = Image.open("screenshot.png")
```

This opens an image file named **screenshot.png**. If the file can't be read, an IOError is raised; otherwise, it returns an instance of class Image. Remember to specify the appropriate path to an image on your system as an argument to the `Image.open` method. The `open()` function determines the format of the file based on its contents.

Now when you have an Image object, you can use the available attributes to examine the file. For example, if you want to see the size of the image, you can call the 'format' attribute.

```
>>> print (im.size, im.mode, im.format)
(1366, 768) RGB PNG
```

The size attribute is a 2-tuple that contains the image's width and height in pixels. RGB is the mode for true colour images. For saving an image, use the `save` method of the Image class. Again, make sure you replace the following string with an appropriate path to your image file.

```
>>> im.save(“changedformat.jpg”)
```

Use the `show` method to view the image. To view the new image, remember to open it up first:

```
>>> newImage = Image.open (“changedformat.jpg”)
>>> newImage.show()
```

Basic image manipulation

So now that we know how to open and save images, let's modify them. The `resize` method works on an image object and returns a new Image object of the specified dimensions. The method accepts a two-integer tuple for the new width and height:

```
>>> width, height = im.size
>>> reducedIM = im.resize((int(width/2), int(height/2)))
>>> reducedIM.save(‘reduced.png’)
>>> reducedIM.show()
```

In the interactive Python shell, we first assign the size and width of the image by extracting them from the tuple into more logically named variables. We then divide these by 2 to reduce both values by half, while calling the `resize` method. The result is an image that's half the size of the original. Note that the `resize` method accepts only integers in its tuple argument, which is why we've wrapped the reduced dimensions in an init() call.

You can similarly change the orientation of the image by rotating it by a specified degree with the rotate method. The rotate method takes a single argument (either integer or float), which is the number of degrees to rotate the image by and, just like resize , returns a new Image object.

```
>>> im.rotate(180).show()
>>> im.rotate(230.5).show()
>>> im.rotate(90).save('OnItsSide.png')
```

The first statement rotates the image by 180 degrees, the second by 230.5 degrees, and the third by 90 degrees. We've saved ourselves some effort by calling show() directly in the first two statements to view the image, and save() in the third to save the rotated image. Note that if the image is rotated to any angle besides 90 or 270, Pillow maintains its original dimensions. When rotated at these two angles, the image swaps its width and height. You can also create a mirror version of the image, using the transpose method:

```
>>> im.transpose(Image.FLIP_LEFT_RIGHT).show()
>>> im.transpose(Image.FLIP_TOP_BOTTOM).show()
```

Also, just like the previous example, replace the call to the show method with the save method, together with the name of a new image file, to save the flipped image. Another common trick is to create thumbnails:

```
>>> size=(128,128)
>>> im.thumbnail(size)
>>> im.show()
>>> im.save('thumbnail.png')
```

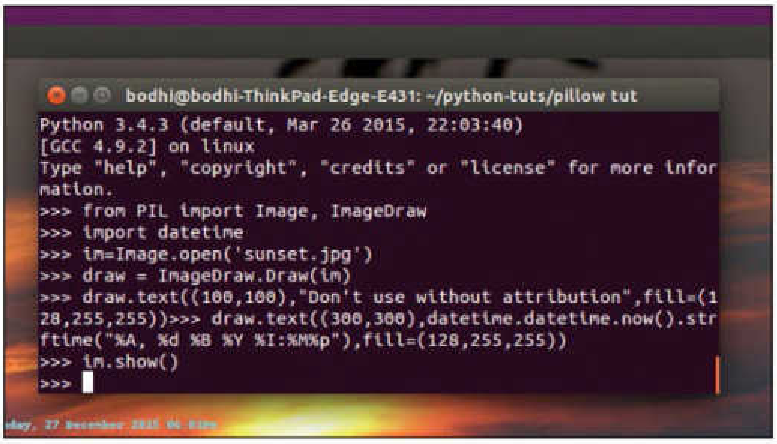
We start by defining a tuple with the dimensions of the thumbnail, and then use it to create the thumbnail in place. Remember that unlike the previous methods, thumbnail changes the original image itself.

In addition to the more common image-editing techniques, Pillow also includes a number of methods and modules that can be used to enhance your images. To begin with, let's take a look at the ImageFilter module, which contains several enhancement filters. To use this module, you will first have to import it with:

```
>>> from PIL import ImageFilter
```

You can now use the various filters of the ImageFilter, such as blur to apply a Gaussian blur effect to the image:

```
>>> im.filter(ImageFilter.BLUR).show()
>>> im.filter(ImageFilter.BLUR).save('blurred.jpg')
```



```
Python 3.4.3 (default, Mar 26 2015, 22:03:40)
[GCC 4.9.2] on linux
Type "help", "copyright", "credits" or "license" for more infor
mation.
>>> from PIL import Image, ImageDraw
>>> import datetime
>>> im=Image.open('sunset.jpg')
>>> draw = ImageDraw.Draw(im)
>>> draw.text((100,100),"Don't use without attribution",fill=(1
28,255,255))>>> draw.text((300,300),datetime.datetime.now().str
ftime("%A, %d %B %Y %I:%M%p"),fill=(128,255,255))
>>> im.show()
>>> 
```

With datetime.datetime.now().strftime("%A, %d %B %Y %I:%M%p") we've used Python's datetime module to timestamp an image.

For more advanced image enhancement, you can use the methods in the ImageEnhance module. The module returns an image object and can be used to make various adjustments, including changes to contrast, sharpness and colour balance, without modifying the original image.

Dressing up images

Begin by importing the ImageEnhance module with:

```
>>> from PIL import ImageEnhance
```

You can now use it to, for example, increase the brightness by 20% with:

```
>>> ImageEnhance.Brightness(im).enhance(1.9).show()
```

Or, perhaps reduce contrast and save the results:

```
>>> newImage=ImageEnhance.Contrast(im).enhance(-1.5)
>>> newImage.save('reducedContrast.jpg')
```

Yet another common image-editing task is cropping portions of an image. Pillow's crop method can help you describe the coordinates of the image you're interested in and discard the rest, such as:

```
>>> width, height = im.size
>>> left = int(width/4)
>>> top = int(height/4)
>>> right = int(3 * width/4)
>>> bottom = int(3 * height/4)
>>> im.crop((left,top,right,bottom)).show()
```

Image processing 101

To manipulate images effectively, you should understand how computers read and display them. The display area of the monitor is made up of coloured dots called pixels. Each pixel represents a colour, and the RGB system is the most common scheme for representing colour. RGB stands for the colour components of red, green and blue, to which the human retina is sensitive. These are mixed together to form a unique colour value. The computer represents these values as integers, which the display hardware translates into the colours we see. Each colour component can range from 0 through 255. The value 255 represents the maximum saturation of a given colour component, whereas the value 0 represents the total absence of that component. So black has an RGB value of 0,0,0, green is 0,255,0, grey is 127,127,127 and yellow is 255,255,0.

The total number of RGB colour values is equal to all the possible combinations of three values: 256 * 256 * 256, or 16,777,216. Because of its wide range of colours, RGB is known as the true colour system. In the old days of black and white monitors, only a single bit of memory was required to represent the two colour values. Early colour monitors could support the display of 256 colours, so 8 bits were needed to represent each colour value. Each colour component of an RGB colour requires 8 bits, so the total number of bits needed to represent a distinct colour value is 24.

When an image is loaded, a computer maps the bits from the file into a rectangular area of coloured pixels. The coordinates range from (0, 0) at the upper-left corner of an image to (width - 1, height - 1) at the lower-right, where width and height are the image's dimensions in pixels. An image consists of a width and a height, accessible by (x, y) coordinates; and a colour value consists of the tuple (r, g, b).

Image.new('RGB', (100, 200), 'white') creates a new white image that's 100 pixels wide and 200 pixels long. You can now draw all kinds of lines and shapes on this new image with Pillow.

```
>>> im.crop((left,top,right,bottom)).save('croppedImage.png')
```

The crop method takes a tuple and returns an Image object of the cropped image. The above statements help cut out the outside edges of the image, leaving you with the centre of the original image. The cropped image's dimensions end up being half of the original image's dimensions. For example, if the image you're cropping is 100 x 100, the left and top variables will both be set to 25, and the right and bottom variables to 75. Thus, you end up with a 50 x 50 image, which is the exact centre of the original image. The int() wrapper makes sure the returned values of the dimensions are integers and not floats.

Then there's the paste method, which as you have probably guessed, pastes an image on top of another one. Remember, however, that the paste method doesn't return an Image object, and instead modifies the image in place. So it's best to make a copy of your image and then call paste on that copy:

```
>>> im=Image.open('screenshot.png')
>>> copyIM=im.copy()
```

Now that we have a copy of our screenshot, we'll first crop a smaller bit from the image, just like we did in the previous section:

```
>>> width, height = im.size
>>> left = int(width/4)
>>> top = int(height/4)
>>> right = int(3 * width/4)
>>> bottom = int(3 * height/4)
>>> croppedIM=copyIM.crop((left,top,right,bottom))
```

The cropped thumbnail image is in **croppedIM**, which we're going to paste over the copied image:

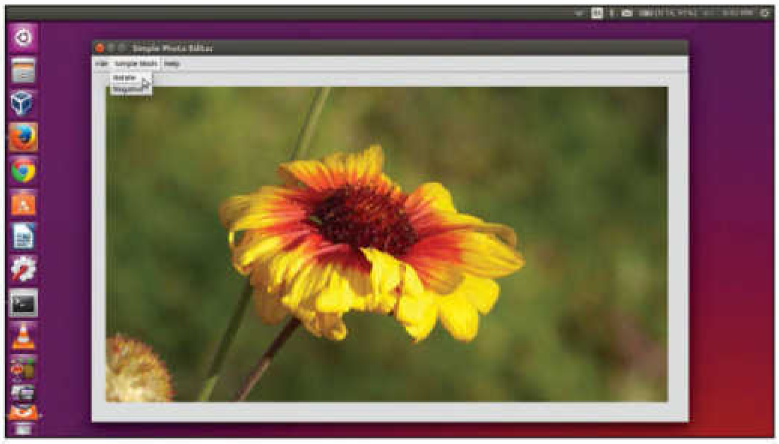
```
>>> copyIM.paste(croppedIM, (400,500)).show()
>>> copyIM.paste(croppedIM, (200,0)).show()
>>> copyIM.save('pastedIM.png')
```

The paste method takes the X and Y coordinates as the two arguments that'll determine the location of the top-left corner of the image being pasted (**croppedIM**) into the main image (**copyIM**). After pasting the thumbnails twice at different locations, we save the image as **pastedIM.png**.

Add a watermark

Perhaps the most popular use of the paste method is to watermark images. With a simple script, you can paste a custom watermark or logo over all the images you've ever shot in a matter of minutes.

```
import os
```



Tkinter is a wonderful toolkit to quickly cook up a lightweight user interface for your Python scripts.

```
from PIL import Image

logoIm = Image.open('logo.png')
logoWidth, logoHeight = logoIm.size

os.makedirs('withLogo', exist_ok=True)
```

Here we've imported the required library, opened our **logo** image and extracted its dimensions into a variable. The last statement creates a directory called **withLogo**, which is where we'll house the logo-fied images.

Next we'll loop over all the images in the current directory with a for loop:

```
for filename in os.listdir('.'):
  if not (filename.endswith('.png') or filename.endswith('.
jpg')):
    continue
```

The additional if statement ignores files that don't end with a .png or .jpg. Modify the statement to include other image formats, or perhaps even drop it if you don't have anything but image files in the directory. Now load the image and extract its size information:

```
im = Image.open (filename)
IMwidth, IMheight = im.size
```

That's all the info we need to paste the logo and save the resulting image in the directory we've just created:

```
print ('Adding the logo to %s' %(filename))
im.paste(logoIm, (IMwidth - logoWidth, IMheight -
logoHeight))
im.save(os.path.join('withLogo', filename))
```

We've used the paste method to place the logo in the bottom-right corner of the image. To determine the coordinates of the top-left corner for placing the logo, we subtract the width and height of the logo image from the real image. In case we want to place the logo in the bottom-left corner, the coordinates will be 0 for the X axis, and the Y axis would be the height of the image minus the height of the logo. In Python terms, it's expressed as:

```
im.paste(logoIm, (0, IMheight - logoHeight))
```

Similarly, we can also place a textual watermark over the images. This, however, is done via the ImageDraw module, which is useful for annotating images. First, modify the script to import additional libraries:

```
from PIL import ImageDraw, ImageFont

draw = ImageDraw.Draw(im)
font = ImageFont.truetype("DroidSans.ttf", 24)
print ("Adding watermark to %s" % (filename))
draw.text((150, 150),"All Rights Reserved",fill=(128,255,128))
```

The first statement creates an object that will be used to draw over the image. The font option specifies which font. PIL can use: bitmap fonts or OpenType/TrueType fonts. To load a OpenType/TrueType font, we use the truetype function in the ImageFont module. Finally, we use the draw.text function to draw a string at the specified position that points to the top-left corner of the text (150x150 in our script). The fill option gives the RGB colour for the text.

Simple image editor

There are several other uses for the Pillow library. You can, for example, use its various functions and methods to modify and apply filters to any image. For ease of use, you can go a step further and wrap the script in a rudimentary interface cooked up with Python's de-facto GUI library Tkinter. Tkinter is covered in detail on page 142, so we won’t go into details about it here. To start constructing our basic image manipulation app, import the required libraries:

```
import tkinter as tk
import tkinter.filedialog
import PIL
from PIL import Image, ImageTk, ImageOps, ImageEnhance,
ImageFilter
import numpy as np
```

Besides Tkinter and PIL, we’ve imported the **NumPy** package for its superior arithmetic skills. Now let’s create the main class that’ll initialise the graphical interface:

```
class ImageEditor(tk.Frame):
  def __init__(self, parent):
    tk.Frame.__init__(self, parent)
    self.parent = parent
    self.initUI()
```

Flip to the Tkinter tutorial on page 142 for a detailed explanation of the methods and functions we’ve used here. The `initUI()` function will draw the actual interface:

```
def initUI(self):
  self.parent.title(“Simple Photo Editor”)
  self.pack(fill = tk.BOTH, expand = 1)
 self.label1 = tk.Label(self, border = 25)
 self.label1.grid(row = 1, column = 1)
 menubar = tk.Menu(self.parent)
 filemenu = tk.Menu(menubar, tearoff=0)
```

The above code gives the app window a title and then uses the Pack geometry manager that helps pack widgets in rows and columns. The `fill` option makes sure the widgets fill the entire space by expanding both horizontally and vertically. The `expand` option is used to ask the manager to assign additional space to the widget box. This will help us resize the window to accommodate larger images. We then create a pull-down menu:

```
filemenu.add_command(label=“Open”, command= self.
onOpen)
filemenu.add_command(label=“Save”)
filemenu.add_command(label=“Close”)
filemenu.add_separator()
filemenu.add_command(label=“Exit”, command=self.quit)
menubar.add_cascade(label=“File”, menu=filemenu)

editmenu = tk.Menu(menubar, tearoff=0)
editmenu.add_command(label=“Rotate”, command=self.
onRot)
editmenu.add_command(label=“Trace Contours”,
command=self.onCont)
menubar.add_cascade(label=“Simple Mods”,
menu=editmenu)

helpmenu = tk.Menu(menubar, tearoff=0)
helpmenu.add_command(label=“Help Index”)
helpmenu.add_command(label=“About...”)
menubar.add_cascade(label=“Help”, menu=helpmenu)

self.parent.config(menu = menubar)
```

The `add_command` method adds a menu item to the menu, `add_seperator` adds a separator line, and the `add_cascade` method creates a hierarchical menu by associating a given menu to a parent menu. Normally, Tkinter menus can be torn off. Since we don’t want that feature, we’ll turn it off by setting `tearoff=0`. We can add menu options as we expand our app. Each menu option has a `command=` option, which executes the associated custom function when the menu item is clicked.

However, before we define the custom functions for editing the images, we have to open the image:

```
def onOpen(self):
  ftypes = [(‘Image Files’, ‘*.jpg *.png, *.gif’)]
  dlg = tkinter.filedialog.Open(self, filetypes = ftypes)
  filename = dlg.show()
  self.fn = filename
  self.setImage()
```

Here we’ve defined the parameters for the Open dialog box to help us select an image. Next comes the `setImage` function that displays the image in our Tkinter interface:

```
def setImage(self):
  self.img = Image.open(self.fn)
  photo = ImageTk.PhotoImage(self.img)
  self.label1.configure(image = photo)
  self.label1.image = photo # keep a reference!
```

Using `PhotoImage` we convert the PIL image into a Tkinter PhotoImage object, so that we can place it on to the canvas, which we do in the next line. Remember to keep a reference to the image object, as we did in the last line. If you don’t, the image won’t always show up.

Moving on, as a nice convenience feature, we can program our editor to resize the image window according to the size of the image we’ve asked it to open. This feature can be added by adding the following bit of code in the `setImage` function before placing the image on to the canvas:

```
self.I = np.asarray(self.img)
l, h = self.img.size
geo = str(l)+“x”+str(h)+“+0+0”
self.parent.geometry(geo)
```

Here we’ve used **NumPy** to help calculate the size of the image being opened and reset the geometry of the window accordingly. The `geometry` method sets a size for the window and positions it on the screen. The first two parameters are the width and height of the window. The last two parameters are x and y screen coordinates.

Now that we have opened and placed an image on the canvas, let’s define a custom function to modify it:

```
def onRot (self):
  im = self.img.rotate(45)  ##rotate an image 45 degrees
  photo2 = ImageTk.PhotoImage(im)
  self.label1.configure(image=photo2)
  self.label1.image = photo2
  self.label1.grid(row=1, column=3)
  self.img = photo2
```

You can similarly create the `onCont` function by replacing the `rotate` command with this:

```
im = self.img.filter(ImageFilter.CONTOUR)
```

As usual, to begin execution, we’ll use Python’s `if __name__ == ‘__main__’ trick`:

```
def main():
  root = tk.Tk()
  ImageEditor(root)
  root.geometry(“300x200”)
  root.mainloop()

if __name__ == ‘__main__’:
  main()
```

The complete script is available on our GitHub at **https://github.com/geekybodhi/techmadesimple-pillow**. Use it as a framework to flesh out the image editor as you learn new tricks and get familiar with the Pillow library. ■

iPad User
Magazine
Do more with your
iPad
iPAD PRO
Worth the upgrade?
Master your tablet's apps and system features

The UK's best-selling Apple mag!
NEW LOOK!
Get more from OS X
El Capitan
Go even deeper into OS X and unlock your Mac's true potential
PLUS! Definitive 4K iMac review
iCloud: how to keep your stuff secure
WIN! An amazing iPhone 6s bundle
MACFORMAT.COM
Mac iPhone iPad Watch iCloud
MACFORMAT › ISSUE 296 › FEBRUARY 2016
TOP TIP! To manually attach a location to a shot in the

ON SALE NOW!
The UK's best-selling Apple mag!
NEW LOOK!
Mac FORMAT
Get more from OS X
El Capitan
YOUR MAC
REBORN
The ultimate guide to making your Mac better than ever!
Revamp OS X › Improve performance › Easy upgrades
NEW SECTION! APPLE HOME
iCloud: how to keep your stuff secure
WIN! An amazing iPhone 6s bundle
Master Apple's App of the Year

PyAudio: Detect and save audio

Discover how Python can be used to record and play back sound.

Audio is an important means of communication and, truth be told, working with multimedia is fun. However, working with audio is a challenging task because it involves handling a variety of devices, standards and formats. There are several libraries that you can use with Python to capture and manipulate sound, and make interactive apps. One of the better platform-independent libraries to make sound work with your Python application is PyAudio. The library can help you record audio as well as play back sounds without writing oodles of code.

The PyAudio library provides Python bindings for the popular cross-platform PortAudio I/O library. It can be easily installed with a simple `pip install pyaudio` command. As with all programs, you need to begin any PyAudio program by importing the library. PyAudio is generally complemented with the WAV library to capture (or play) audio in the lossless WAV format:

```
import pyaudio
import wave
```

Capture audio

Once you've imported the libraries, you can quickly cobble together the code to open the PyAudio recording stream. While initialising the stream, we'll also define a number of properties that can be altered to influence how the data is captured and recorded.

```
p = pyaudio.PyAudio()
stream = p.open (format=pyaudio.paInt16,
                channels=1,
                rate=44100,
                input= True,
                input_device_index = 0,
                frames_per_buffer = 1024)
```

The pyaudio.PyAudio class is used to initiate and terminate PortAudio, access audio devices, and open and close audio streams. The real audio processing happens in the pyaudio.Stream class, where the stream is opened with several parameters. The `pyaudio.paInt16` parameter sets the bit depth to 16 bits, which indicates that you are reading the data as 16-bit integers; `channels` can be set to 1 for mono or 2 for stereo; `rate` defines the sampling frequency, which is usually 44.1 kHz. The `input=True` parameter asks PyAudio to use the stream as input from the device, which is mentioned with the `input_device_index` parameter.

`Frames_per_buffer` is an interesting parameter, which we use to define the length of the audio buffer. The data captured in this buffer can then either be discarded or saved. PyAudio uses chunks of data, instead of a continuous amount of audio, for a couple of reasons. For one, reading data in chunks helps use resources efficiently. Recording a continuous flow of data from the microphone would just eat up the processor and may lead to memory leaks on devices such as the Raspberry Pi, which has a limited amount of RAM. Another advantage of reading data in chunks, as we will see later on, is that it enables us to analyse the data and only save those relevant bits that meet our criteria, and discard the rest.

In a real program, you'll want to define these parameters in variables, because they'll be used in other places in the program as well. Along with the above, we will also define two variables:

```
record_seconds = 10
output_filename = 'Monoaudio.wav'
```

Now comes the code that does the actual audio capture:

```
print (" Now recording audio. Speak into the microphone. ")
frames = [ ]

for i in range(0, int(rate / chunk * record_seconds)):
    data = stream.read(chunk)
    frames.append(data)
```

We first print a line of text that notifies the start of the recording, and then define a list variable named `frames`. The `for` loop handles the task of recording the audio. It brings in the data in correctly sized chunks and saves it into a list named `frames`. The `for` loop helps determines how many samples to record given the rate, the number of bytes a recording should have, and the recording duration. In our case, the loop makes 430 iterations and saves the captured audio data in the frames list. We will now close the audio recording stream:

```
print (" Done recording audio. ")
stream.stop_stream()
stream.close()
p.terminate()
```

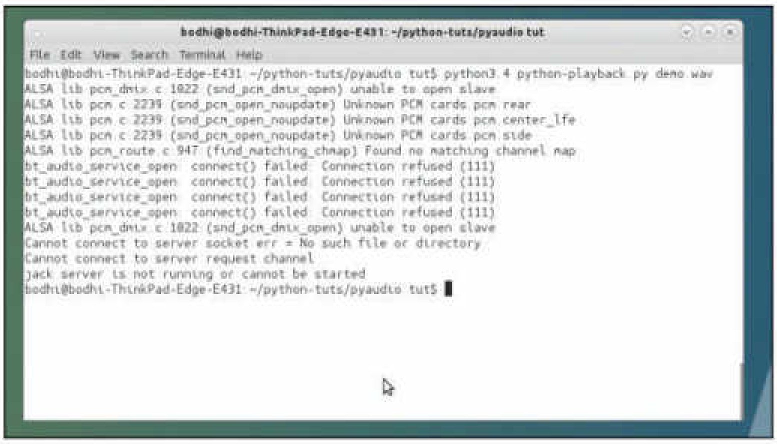
The above code first terminates the audio recording, then closes the stream, and finally terminates the PyAudio session. Now that we have the audio data as a string of bytes in the frames list, we'll transform it and save it as a WAV file using the wave module:

```
wf = wave.open(output_filename, 'wb')
wf.setnchannels(channels)
wf.setsampwidth(p.get_sample_size(format))
wf.setframerate(rate)
wf.writeframes(b' '.join(frames))
wf.close()
```

This is where the data is actually written to an audio file. Python opens the file in the write-only mode, as defined by the wave module. This is followed by a bunch of wave write objects, such as setnchannels, which sets the number of channels (mono in our case), and setframerate, which sets the frame rate (44.1kHz in our case). The setsamwidth object sets the sample width and is derived from the format using PyAudio's `get_sample_size()` function. Finally, the `b' '.join()` method combines the bytes in a list, which in our case is named frames, before closing the file.

If you run the program, you'll first notice a string of warnings. You can safely ignore these because they are the result of PyAudio's verbose interactions with the audio hardware on your computer. What you should watch out for are any Python exceptions that point to actual errors in the code and must be ironed out. After executing the program, you'll have a 10-second WAV file named **Monoaudio.wav** in the current directory.

While the above code will work flawlessly and helps demonstrate the ease of use of the PyAudio library, this isn't how you'd program it in the real world. In a real project, you'd break up the tasks into various smaller tasks and write a class for each. In such a case, the code that begins the execution would look something like:



```
bodhi@bodhi-ThinkPad-Edge-E431: ~/python-tuts/pyaudio tut
```

```
bodhi@bodhi-ThinkPad-Edge-E431 ~/python-tuts/pyaudio tut$ python3.4 python-playback.py demo.wav
ALSA lib pcm_dmix.c 1022 (snd_pcm_dmix_open) unable to open slave
ALSA lib pcm.c 2239 (snd_pcm_open_noupdate) Unknown PCM cards.pcm.rear
ALSA lib pcm.c 2239 (snd_pcm_open_noupdate) Unknown PCM cards.pcm.center_lfe
ALSA lib pcm.c 2239 (snd_pcm_open_noupdate) Unknown PCM cards.pcm.side
ALSA lib pcm_route.c 947 (find_matching_chmap) Found no matching channel map
bt_audio_service_open  connect() failed  Connection refused (111)
bt_audio_service_open  connect() failed  Connection refused (111)
bt_audio_service_open  connect() failed  Connection refused (111)
bt_audio_service_open  connect() failed  Connection refused (111)
ALSA lib pcm_dmix.c 1022 (snd_pcm_dmix_open) unable to open slave
Cannot connect to server socket err = No such file or directory
Cannot connect to server request channel
jack server is not running or cannot be started
bodhi@bodhi-ThinkPad-Edge-E431 ~/python-tuts/pyaudio tut$ ▌
```

› You can safely ignore all messages (except Exception errors) that appear whenever you run a PyAudio script – it's simply the library discovering the audio hardware on your machine.

```
if __name__ == '__main__':
  rec = Recorder(channels=1)
  with rec.open('capture.wav', 'wb') as recfile:
    recfile.record(duration=10.0)
```

We discuss Python's `if __name__ == '__main__'` trick for writing reusable code in the Tweepy tutorial *(see page 138)*. The above code is executed when the program is run directly. Now, when the `with` statement is executed, Python evaluates the expression, and calls the magic `__enter__` method on the resulting value (which is called a context guard), and assigns whatever `__enter__` returns to the variable given by `as`. Python then executes the code body and, irrespective of what happens in that code, it calls the guard object's `__exit__` method. The `__enter__` and `__exit__` methods make it easy to build code that needs some setup and shutdown code to be executed – for example, to close the file in our case:

```
def __exit__(self, exception, value, traceback):
  self.close()
```

The Recorder class opens a recording stream and defines its parameters:

```
class Recorder(object):
  def __init__(self, channels=1, rate=44100, frames=1024):
    self.channels = channels
    self.rate = rate
```

»

Managing a PyAudio stream

We've used some of the parameters that you can use with the pyaudio.Stream class. Here's a list of the other popular ones. The stream, which can either be an input stream, an output stream or both, is initialised with the `__init__` function. You can pass several parameters to the stream while creating it, as we've done. The popular ones are the sampling rate, the number of channels, the number of frames per buffer, and the sampling size and format. In the tutorial, we've used paInt16 format, which sets the bit depth to 16 bits, indicating that you are reading the data as 16-bit integers. In addition to this, other formats include paInt8, paInt24, paInt32 and paFloat32. Audiophile developers are advised to spend some time with the PortAudio documentation to understand the nuances of these different formats before selecting one for their application. Moving on, there are several other parameters that can be used to fetch information from a sound file while initialising a stream. The `get_input_latency()` function returns the float value of the input latency, while `get_time()` returns the stream's time. For managing the stream, besides the `start_stream()` and `stop_stream()` functions, there's also the `is_active()` and `is_stopped()` functions, which return a Boolean value depending on whether the stream is active or stopped.

```
»     self.frames_per_buffer = frames

   def open(self, fname, mode='wb'):
      return RecordingFile (fname, mode, self.channels, self.rate,
 self.frames)
```

The class defines the necessary parameters and then passes them to another class, named RecordingFile, which defines the record function that opens the stream, captures the audio, and then writes the files using the `for` loop, as described earlier.

Vocalise recordings

You can easily adapt the code to play back the file. In addition to pyaudio and wave, we'll import the sys module for accessing system-specific functions and parameters.

```
import pyaudio
import wave
import sys

if len(sys.argv) < 2:
   print(" The script plays the provided wav file.\n
                   Proper usage: %s filename.wav" % sys.
argv[0])
   sys.exit(-1)
```

The script will play any provided WAV file. If one isn't provided, the above code comes into action and prints the proper usage information before exiting.

```
wf = wave.open(sys.argv[1], 'rb')
p = pyaudio.PyAudio()
stream = p.open(format=p.get_format_from_width(wf.
getsampwidth()),
            channels=wf.getnchannels(),
            rate=wf.getframerate(),
            output=True)
```

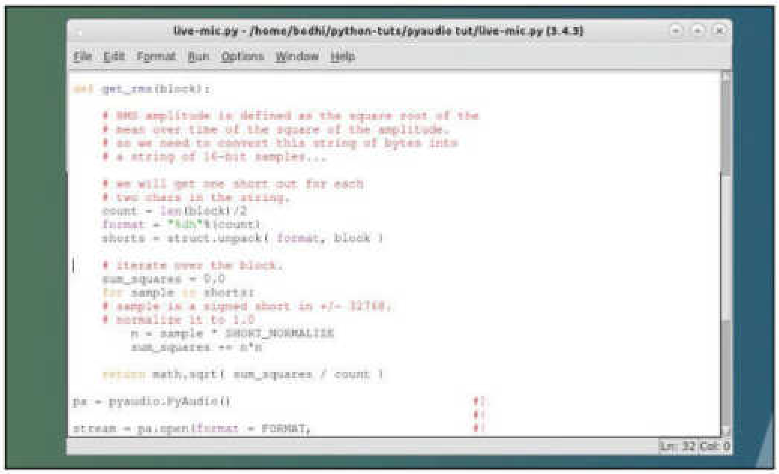
If a filename is provided, it's opened in read-only mode and its contents are transferred to a variable named **wf**. Then the PyAudio library is initialised. Unlike the `open()` function we used earlier to define the parameters of the stream, when reading a WAV file, we'll use functions to extract details from the file. For instance, the `getsampwidth` method fetches the sample width of the file in bytes. Similarly, the `getnchannels` method returns the number of audio channels, and `getframerate` returns the sampling frequency of the wave audio file.

```
# read data
data = wf.readframes(1024)

# play stream (3)
while len(data) > 0:
   stream.write(data)
   data = wf.readframes(1024)
```

Once we have all the details from the file, we extract the frames in chunks of 1,024 bytes. As long as there are chunks of data, the `while` loop writes the current frame of audio data to the stream. When all the frames have been streamed, we'll then terminate the stream and close PyAudio with:

```
stream.stop_stream()
stream.close()
p.terminate()
```

› For more accurate results, calculate the RMS value for a group of sound frames to decide what level is considered to be silence.



live-mic.py - /home/bodhi/python-tuts/pyaudio tut/live-mic.py (3.4.3)

```
def get_rms(block):

    # RMS amplitude is defined as the square root of the
    # mean over time of the square of the amplitude.
    # so we need to convert this string of bytes into
    # a string of 16-bit samples...

    # we will get one short out for each
    # two chars in the string.
    count = len(block)/2
    format = "%dh"%(count)
    shorts = struct.unpack( format, block )

    # iterate over the block.
    sum_squares = 0.0
    for sample in shorts:
    # sample is a signed short in +/- 32768.
    # normalize it to 1.0
        n = sample * SHORT_NORMALIZE
        sum_squares += n*n

    return math.sqrt( sum_squares / count )

pa = pyaudio.PyAudio()                                   #|
                                                         #|
stream = pa.open(format = FORMAT,                        #|
```

Auto recorder

In the scripts above, we've seen how to use PyAudio to record from the microphone for a predefined number of seconds. With a little bit of wizardry, you can also train PyAudio to start recording as soon as it detects a voice, and keep capturing the audio until we stop talking and the input stream goes silent once again.

To do this, we'll initiate the script as usual with the `if __name__ == '__main__':` trick to print a message and then call the function to record the audio, such as:

```
print("Start talking into the mic")
record_to_file('capture.wav')
print("Alrighty, we're done. Recorded to capture.wav")
```

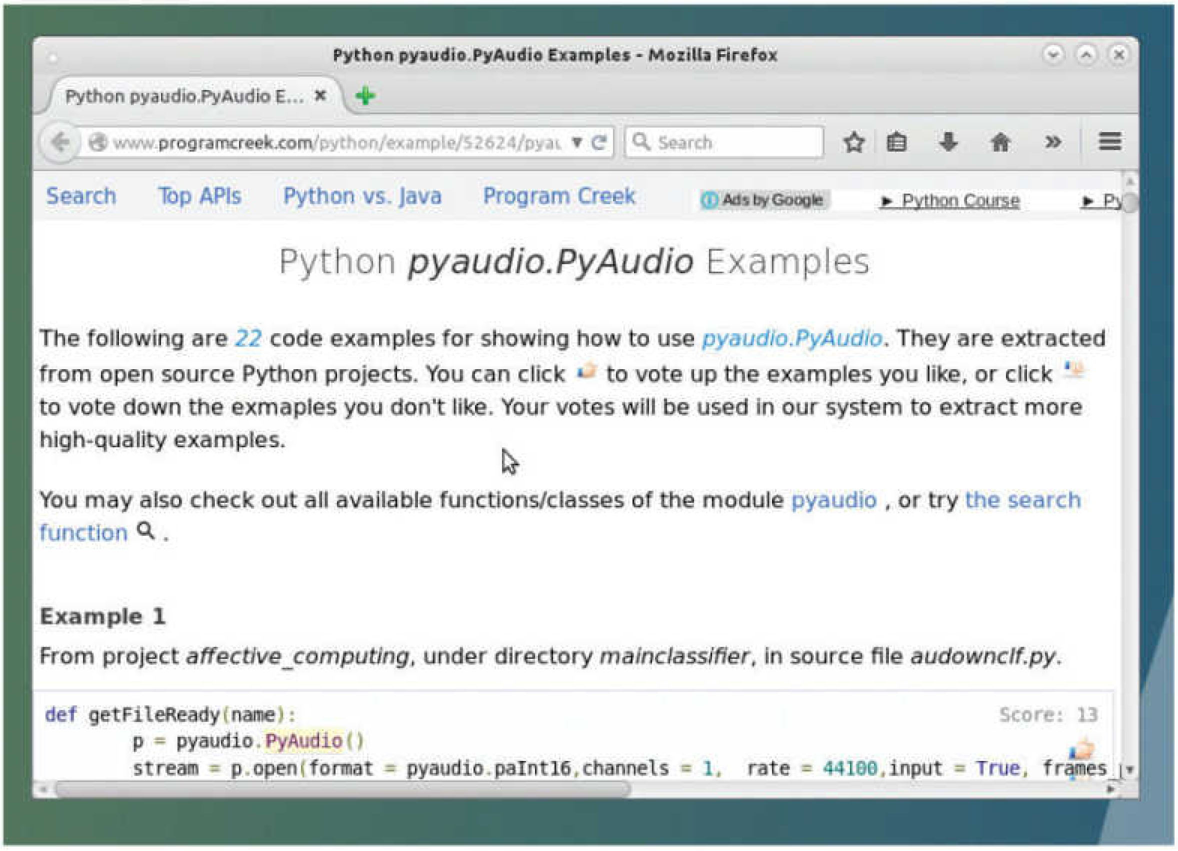
The `record_to_file` function receives the name of the file to record the sound data to, and contains the code to record data from the microphone into the file after setting up the stream parameters. It also calls the `record()` function, which

Other popular Python sound libraries

Python has some interesting built-in multimedia modules for working with sound, as well as a bunch of external ones. We used the wave module to read and write audio files in the WAV audio format. There's also the built-in audioop module, used for manipulating the raw audio data. It can be used to perform several operations on sound fragments. For example, you can find the minimum and maximum values of all the samples within a sound fragment.

There are also several third-party open source multimedia frameworks with Python bindings. PyMedia is a popular open source media library that supports audio/video manipulation of a wide range of multimedia formats including WAV, MP3, OGG, AVI, DivX, DVD and more. It enables you to parse, demutiplex, multiplex, decode and encode all supported formats. There's also Pydub, which is a high-level audio interface that uses FFMPEG to support formats other than WAV. The GStreamer Python bindings allow developers to use the Gstreamer framework to develop applications with complex audio/video processing capabilities, from simple audio playback to video playback, recording, streaming and editing.

One of the best features of the Pyglet multimedia and windowing library is that it needs nothing else besides Python, simplifying installation. It's popular for developing games and other visually rich applications. It can also work with AVbin to play back audio formats such as MP3, OGG/Vorbis and WMA, and video formats such as DivX, MPEG-2, H.264, WMV and Xvid. Another popular library is Pygame, which adds functionality on top of the SDL library.


Python pyaudio.PyAudio Examples - Mozilla Firefox
Python pyaudio.PyAudio E...
www.programcreek.com/python/example/52624/pyau
Search
Search
Top APIs
Python vs. Java
Program Creek
Ads by Google
► Python Course
► Py

Python *pyaudio.PyAudio* Examples

The following are 22 code examples for showing how to use pyaudio.PyAudio. They are extracted from open source Python projects. You can click to vote up the examples you like, or click to vote down the exmaples you don't like. Your votes will be used in our system to extract more high-quality examples.

You may also check out all available functions/classes of the module pyaudio , or try the search function .

Example 1

From project *affective_computing*, under directory *mainclassifier*, in source file *audowncIf.py*.

```
def getFileReady(name):                                                   Score: 13
        p = pyaudio.PyAudio()
        stream = p.open(format = pyaudio.paInt16,channels = 1,  rate = 44100,input = True, frames
```

Websites such as Program Creek help you find sample code for Python modules like PyAudio.

among other things, checks the captured data for silence.

```
def record():
    p = pyaudio.PyAudio()
    stream = p.open(format=FORMAT, channels=1, rate=RATE,
                    input=True,frames_per_buffer=CHUNK_
SIZE)

    num_silent = 0
    snd_started = False

    r = array('h')

    while 1:
        snd_data = array('h', stream.read(CHUNK_SIZE))

        if byteorder == 'big':
            snd_data.byteswap()
            r.extend(snd_data)
```

The function records words from the microphone and returns the data as an array of signed shorts. We next check whether this recorded data contains silence by calling the `is_silent` function with `silent = is_silent(snd_data)`. We've defined the `is_silent (snd_data)` function elsewhere in the script. It uses the `max(list)` method, which returns the elements from a list (**snd_data** in this case) with the maximum value. This function will return **True** if the `snd_data` is below the silence threshold. The threshold intensity defines the silence-to-noise signal. A value lower than the threshold intensity is considered silence. Once the `is_silent` function returns a value (either **True** or **False**), we'll then subject it to further tests:

```
if silent and snd_started:
    num_silent += 1
elif not silent and not snd_started:
    snd_started = True

if snd_started and num_silent > 100:
    break
```

If the captured data is above the threshold of silence and we haven't been recording yet, we'll set the `snd_started` variable to **True** to begin recording sound. But if we've been recording sound and the captured data is below the silence threshold, we'll increment the `num_silent` variable. Moving on, we'll stop recording if we've been recording but it's been 100 frames since the last word was spoken.

Next, we'll call another function to manipulate the recording and pad the audio with 1.5 seconds of blank sound. Adding silence is simple. We'll just read the recorded data into an array and then extend it by adding blank frames.

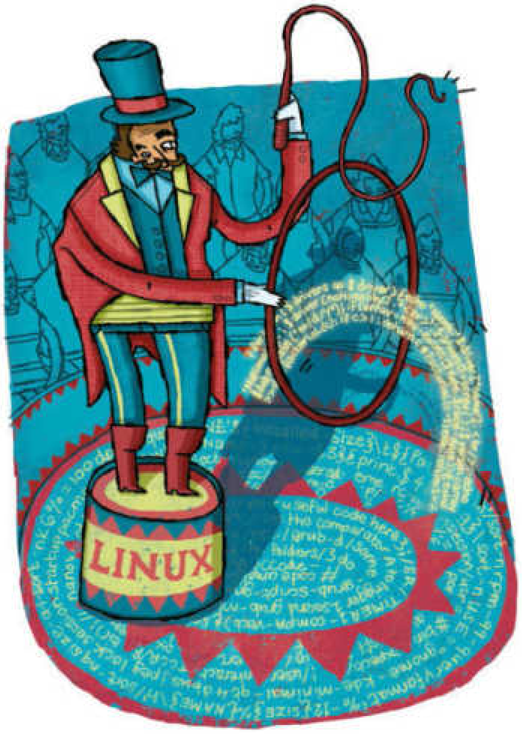
```
def add_silence(snd_data, seconds):
    r = array('h', [0 for i in range(int(seconds*RATE))])
    r.extend(snd_data)
    r.extend([0 for i in range(int(seconds*RATE))])
    return r
```

The complete script is on our GitHub at **https://github.com/geekybodhi/techmadesimple-pyaudio/**. When you execute it, it will detect and start capturing audio frames but discard them until it detects a voice. From here on, it'll save all consequent frames until it detects a period of silence.

The script also contains comments to help you extend it by writing a `trim` function that'll trim the silent portion from the end and even from the start, if it exists. Although there are several other Python libraries for playing sound, PyAudio is one of the easiest to use with third-party audio toolkits that you can use to capture and manipulate sound, so have fun playing around with your sound! ■

Tweepy: Create Twitter trends

With some Python know-how, you can tap into the collective consciousness and extract "valuable" information from it.



Twitter is a wonderful tool if you want to get a feel for what the world is thinking. The visible 140-character social network is powered by a wonderful API that enables users to drink from the firehose and capture all the raw data passing over the network. Once captured, you can use this data to mine all kinds of meaningful information. You can, for example, find trends related to specific keywords or gauge sentiments during events.

Tweepy is one of the easiest libraries that you can use to access the Twitter API with Python. In this tutorial, we'll be using Tweepy to capture any tweets that mention OggCamp, and then we'll mine the data to print a list of the most frequently used hashtags.

You can easily install Tweepy with `pip install tweepy`. Tweepy accesses Twitter via the OAuth standard for authorisation. For that, you'll first have to head over to **http://apps.twitter.com** and sign in with your regular Twitter credentials. Now press the Create New App button and fill in all the fields except for the Callback URL, which you can leave blank, and then click on the Create button. Once the app has been created, you can hop over to the Keys and Access Tokens tab and create an OAuth token. This might take a minute or two. Once it's done, make a note of the strings listed alongside API Key, API Secret, Access Token and Access Token Secret.

You're now all set to access Twitter with Python. To access the tweets, you need to import the following libraries in your Python script:

```
import tweepy
from tweepy.streaming import StreamListener
from tweepy import OAuthHandler
from tweepy import Stream
```

Next you need to paste in the API and Access tokens, and use these to connect to Twitter, such as:

```
consumer_key = "ENTER YOUR API KEY"
consumer_secret = "ENTER YOUR API SECRET"
access_token = "ENTER YOUR ACCESS TOKEN"
access_secret = "ENTER YOUR ACCESS TOKEN SECRET"

auth = OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_secret)

api = tweepy.API(auth)
```

From this point forward, we can use the `api` variable for interacting with Twitter. For example, you can read your own time with the following loop:

```
for status in tweepy.Cursor(api.user_timeline).items():
  print(status.text)
```

The code will print all status updates in groups of 20 from your own timeline. We use Tweepy's Cursor interface, which supports several different types of objects. Here we have used the `items` method to iterate through our timeline. To limit the number of tweets you wish to print, you can also change the statement to something like `tweepy.Cursor(api.user_timeline).items(10)`, which will only print the 10 most recent tweets.

Capture tweets

As you can see, interacting with Twitter via Python is reasonably straightforward. Twitter also offers a set of streaming APIs, however, and by using these, you can access Twitter's global stream of tweets. The public streams work really well for following specific topics and mining data. We'll import and modify Tweepy's StreamListener class to fetch the tweets.

```
from tweepy import Stream
from tweepy.streaming import StreamListener
from tweepy import OAuthHandler

```

Quick tip

Scroll through Tweepy's documentation (**http://docs.tweepy.org/en/latest/api.html**) for a list of supported methods.

```
auth = tweepy.OAuthHandler(consumer_key, consumer_
secret)
auth.set_access_token(access_token,access_secret)
api = tweepy.API(auth)

class StdOutListener(StreamListener):

  def on_status(self, data):
        print(data)
        return True

  def on_error(self, status):
        print(status)
          return False
```

The Twitter streaming API is used to download Twitter messages in real time. It is useful for obtaining a high volume of tweets, or for creating a live feed using a site stream or user stream. Tweepy establishes a streaming session and routes messages to the StreamListener instance.

Here we begin by creating a class that inherits from StreamListener. StreamListener has several methods. Of these, `on_data()` , `on_status()` and `on_error()` are the most useful ones. The `on_data()` method handles replies to statuses, warnings, direct messages and so on, and passes data from status to the `on_status()` method.

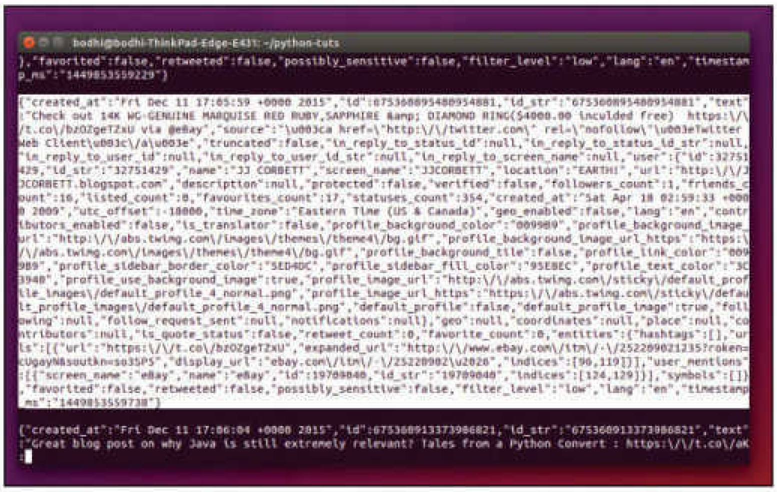
When you're interacting with Twitter via its API, you have to be mindful of rate limiting. If you exceed the limited number of attempts to connect to the streaming API within a window of time, the connect will error out. Repeated attempts will only aggravate Twitter and it will exponentially increase your wait time. To avoid such a situation, use Tweepy's `on_error()` method to print the error messages and terminate execution.

```
listener = StdOutListener()
twitterStream = Stream(auth, listener)
```

Once we've authenticated with Twitter, we can now start the stream listener. The modified StreamListener class is used to create a listener instance. This contains the information about what to do with the data once it comes back from the Twitter API call.

```
if __name__ == '__main__':
  twitterStream.filter(track=["oggcamp"])
```

Here we use our stream class to filter the Twitter stream to fetch only tweets that contain the word 'oggcamp'. The track parameter is an array of search terms to stream. We place this inside an if condition. The `if __name__ == "__main__":` trick exists in Python, so our Python files can act as reusable modules or as standalone programs. Every module has a name, and statements in a module can find out the name of its module. When the Python interpreter reads a source file, it executes all the code found in it. Before executing the code, it defines a few special variables. For example, if the Python interpreter is running the source file as the main program, it sets the `special __name__` variable to have a value `"__main__"` . If this file is being imported from another module, `__name__` is set to the module's name. In our case, as the defined `__name__` is `'__main__'` it tells Python the module is being run standalone by the user and we can do corresponding appropriate actions.



bodhi@bodhi-ThinkPad-Edge-E431: ~/python-tuts

```
},"favorited":false,"retweeted":false,"possibly_sensitive":false,"filter_level":"low","lang":"en","timestam
p_ms":"1449853559229"}

{"created_at":"Fri Dec 11 17:05:59 +0000 2015","id":675360895480954881,"id_str":"675360895480954881","text"
:"Check out 14K WG-GENUINE MARQUISE RED RUBY,SAPPHIRE &amp; DIAMOND RING($4000.00 inculded free)  https:\/\
/t.co\/bzOZgeTZxU via @eBay","source":"\u003ca href=\"http:\/\/twitter.com\" rel=\"nofollow\"\u003eTwitter
Web Client\u003c\/a\u003e","truncated":false,"in_reply_to_status_id":null,"in_reply_to_status_id_str":null,
"in_reply_to_user_id":null,"in_reply_to_user_id_str":null,"in_reply_to_screen_name":null,"user":{"id":32751
429,"id_str":"32751429","name":"JJ CORBETT","screen_name":"JJCORBETT","location":"EARTH!","url":"http:\/\/J
JCORBETT.blogspot.com","description":null,"protected":false,"verified":false,"followers_count":1,"friends_c
ount":16,"listed_count":0,"favourites_count":17,"statuses_count":354,"created_at":"Sat Apr 18 02:59:33 +000
0 2009","utc_offset":-18000,"time_zone":"Eastern Time (US & Canada)","geo_enabled":false,"lang":"en","contr
ibutors_enabled":false,"is_translator":false,"profile_background_color":"009989","profile_background_image_
url":"http:\/\/abs.twimg.com\/images\/themes\/theme4\/bg.gif","profile_background_image_url_https":"https:\
/\/abs.twimg.com\/images\/themes\/theme4\/bg.gif","profile_background_tile":false,"profile_link_color":"009
989","profile_sidebar_border_color":"5ED4DC","profile_sidebar_fill_color":"95E8EC","profile_text_color":"3C
3940","profile_use_background_image":true,"profile_image_url":"http:\/\/abs.twimg.com\/sticky\/default_prof
ile_images\/default_profile_4_normal.png","profile_image_url_https":"https:\/\/abs.twimg.com\/sticky\/defau
lt_profile_images\/default_profile_4_normal.png","default_profile":false,"default_profile_image":true,"foll
owing":null,"follow_request_sent":null,"notifications":null},"geo":null,"coordinates":null,"place":null,"co
ntributors":null,"is_quote_status":false,"retweet_count":0,"favorite_count":0,"entities":{"hashtags":[],"ur
ls":[{"url":"https:\/\/t.co\/bzOZgeTZxU","expanded_url":"http:\/\/www.ebay.com\/itm\/-\/252209021235?roken=
cUgayN&soutkn=so35P5","display_url":"ebay.com\/itm\/-\/25220902\u2026","indices":[96,119]}],"user_mentions"
:[{"screen_name":"eBay","name":"eBay","id":19709040,"id_str":"19709040","indices":[124,129]}],"symbols":[]}
,"favorited":false,"retweeted":false,"possibly_sensitive":false,"filter_level":"low","lang":"en","timestamp
_ms":"1449853559738"}

{"created_at":"Fri Dec 11 17:06:04 +0000 2015","id":675360913373986821,"id_str":"675360913373986821","text"
:"Great blog post on why Java is still extremely relevant? Tales from a Python Convert : https:\/\/t.co\/aK
:
```

All that gobbledygook is just one tweet!

You can modify the above to filter your results based on multiple keywords. For instance, using `twitterStream.filter(track=['LXF', 'tuxradar', 'LinuxFormat'])` will get tweets that contain any of the three words specified: 'LXF', 'tuxradar' or 'LinuxFormat'.

Similarly, if you wish to follow a particular Twitter account, you'll first have to find out its Twitter ID using a web service such as Twitterid.com. Once you have discovered the Twitter ID, you can modify the above to say `twitterStream.filter(follow=['255410383'])` , which will display any new tweet by @LinuxFormat. You can also filter results by combining »

Anatomy of a tweet

The 140 characters of the tweet that are visible on your timeline are like the tip of the iceberg. A complete tweet in its JSON form contains a lot more information and attributes in addition to the tweet itself. A knowledge of the important ones will help you extract the relevant data.

The JSON string begins with the created_at field, which lists the UTC time when the tweet was created. It is followed by the id and the id_str fields, which are the integer and string representation of the user who has contributed to the tweet. Then comes the text field, which we've used extensively in the tutorial – this contains the actual text of the status update. We've also use the entities field, which contains information that has been parsed out of the text of the tweet, such as URLs and hashtags.

Besides these, there's the favorited and retweeted fields, Boolean fields that indicate whether the user whose credentials are being used (you) has marked the tweet as favourite or retweeted it. Then there are the favorite_count and retweet_count fields, which contain the number of times the tweet has been favourited and retweeted respectively.

Also important is the lang field, which contains the two-character language identifier of the tweet, such as en or de. The place and geo fields contain information that points to the geographic location of the tweet.

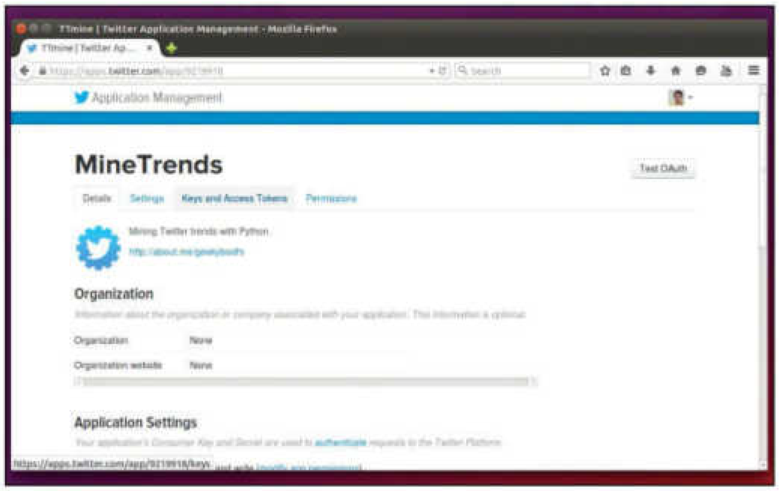
» various parameters. For example, `twitterStream.filter(location=[51.5073509,-0.12775829999998223], track=['linux'])` will fetch any tweets that originate in London and contain the word 'linux'. Take a look at Twitter's official API documentation (**https://dev.twitter.com/streaming/overview/request-parameters**) for a list of all the supported streaming request parameters.

The StreamListener will remain open and fetch tweets based on the specified criteria until you ask it to stop by terminating the script. The large blobs of text on your screen are the tweets fetched by the streamlistener in the JSON format (see box below).

Process tweets for frequency

Before you can process the tweets, you will have to save them to a file.

```
import io

class StdOutListener(StreamListener):

 def __init__(self):
  self.tweet_data=[]
```



› **Head over to http://apps.twitter.com and register a new app to get yourself some API keys and access tokens.**

```

 def on_data(self, data):
  self.tweet_data.append(data)
  saveFile = io.open('oggcampTweets.json', 'w',
encoding='utf-8')
  saveFile.write(u'[\n')
  saveFile.write(','.join(self.tweet_data))
  saveFile.write(u'\n]')
  saveFile.close()
  return True
```

Instead of printing the tweets to the screen, the above block will import the io library and use it to append the data flowing from the StreamListener into a file named **oggcampTweets.json**. It opens the output file, writes the opening square bracket, writes the JSON data as text separated by commas, then inserts a closing square bracket, and closes the document.

Alternatively, you can simply redirect the output of the StreamListener to a file. For example, `python fetch_oggcamp.py > oggcampTweets.json` will save all tweets with the word 'oggcamp' to a file named **oggcampTweets.json**.

We can now use the **oggcampTweets.json** file to mine all kinds of data. For example, let's analyse this data to list the 10 most frequently used hashtags. Begin by importing the relevant libraries:

```
import sys
import json
import operator
```

Now we'll pass the .json file to the script from the command line.

```
if __name__ == '__main__':
 if len(sys.argv) == 2:
  main(sys.argv[1])
 else:
  print('Usage: python top_tweets.py file-with-tweets.json')
```

The following code analyses the number of parameters passed from the command line. If it finds a pointer to the JSON containing the tweets file, it's forwarded to the `main()` function. Else the script prints the proper usage information if the user has forgotten to point to the JSON file.

```
def main(tweetsFile):
 tweets_file = open(tweetsFile)
```

Save tweets to database

Instead of collecting tweets inside a JSON file, you can also directly connect to and save them inside a MySQL database. The MySQL database provides a connector for Python 3 that you can easily install by using `pip3 install mysql-connector-python`. After installing the connector, you can create a database and table for saving the tweets with:

```
CREATE DATABASE tweetTable;
USE tweetTable;
CREATE TABLE tweets (username
VARCHAR(15),tweet VARCHAR(140));
```

This creates a database called **tweetTable** with a table named **tweets** that has two fields to store the username and the tweet. You can now use the following code to import the required Python libraries and connect to the database:

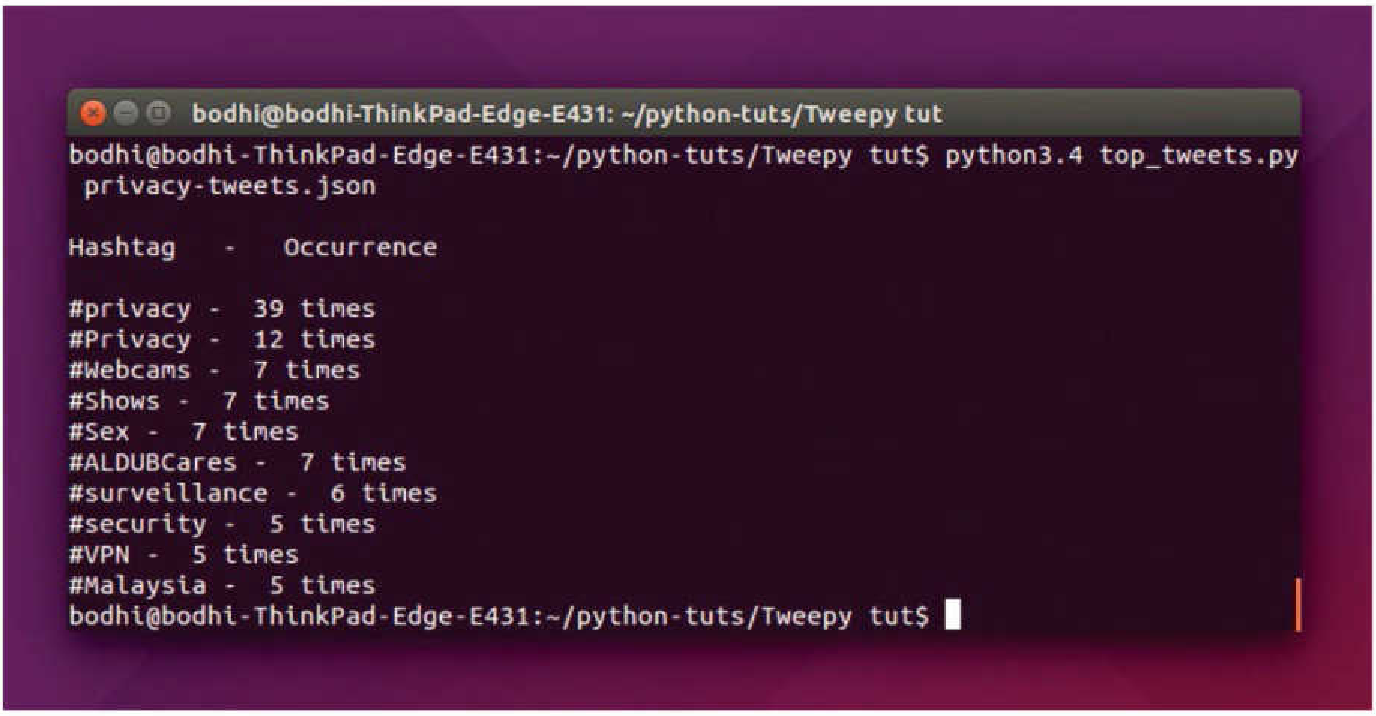
```
import mysql.connector
import json

connect = mysql.connector.
connect(user='MySQLuser',
password='MySQLpassword', host='localhost',
database='tweetTable')
db=connect.cursor()
```

Once we're connected, we'll create a StreamListener, just like we have done in the main tutorial. We'll then scan each tweet for text, and extract the tweet along with the username and screen name from the tweets:

```
class StdOutlistener(StreamListener):
 def on_data(self, data):
  all_data = json.loads(data)
  if 'text' in all_data:
   tweet = all_data["text"]
   username = all_data["user"]["screen_name"]
```

Once we've extracted the information from a tweet in the stream, we can write these to the table, and also print the values on the screen.

```
db.execute("INSERT INTO tweets (username,
tweet) VALUES (%s,%s)",(username, tweet))
connect.commit()

print((username,tweet))
return True
```



```
bodhi@bodhi-ThinkPad-Edge-E431: ~/python-tuts/Tweepy tut
bodhi@bodhi-ThinkPad-Edge-E431:~/python-tuts/Tweepy tut$ python3.4 top_tweets.py
 privacy-tweets.json

Hashtag   -   Occurrence

#privacy -  39 times
#Privacy -  12 times
#Webcams -  7 times
#Shows -  7 times
#Sex -  7 times
#ALDUBCares -  7 times
#surveillance -  6 times
#security -  5 times
#VPN -  5 times
#Malaysia -  5 times
bodhi@bodhi-ThinkPad-Edge-E431:~/python-tuts/Tweepy tut$ 
```

› These are the top hashtags that people used while tweeting about privacy.

```
tweets_hash = {}
for tweet_line in tweets_file:
  if tweet_line.strip():
    tweet = json.loads(tweet_line)
    if "entities" in tweet.keys():
      hashtags = tweet["entities"]["hashtags"]
      for ht in hashtags:
        if ht != None:
          if ht["text"].encode("utf-8") in tweets_hash.keys():
            tweets_hash[ht["text"].encode("utf-8")] += 1
          else:
            tweets_hash[ht["text"].encode("utf-8")] = 1
```

The above might look like a mouthful but it's actually really simple. To start with, the JSON file is received as the `tweetsFile` variable, which is then read by the `tweets_file` variable. We then define a dictionary variable named `tweets_hash` that will house the hashtags and their occurrence frequencies. We then initiate a loop for every individual tweet in the file. We'll first convert the JSON string into a dictionary object, and save it in a variable named `tweet`. If the tweet has a tag named 'entities', we'll extract the hashtag value. Next, we'll check whether the hashtag is already listed in our dictionary. In case it is, we'll just increment its occurrence frequency. Else, we'll add it to the dictionary and record the occurrence.

```
sortedHashTags = dict(sorted(tweets_hash.items(),
key=operator.itemgetter(1), reverse=True)[:10])

print("\nHashtag  -   Occurrence\n")
for count,value in sorted(sortedHashTags.items(),
key=lambda kv: (kv[1],kv[0]),reverse=True):
print("#%s -  %d times" % (count.decode("utf-8"), value))
```

You'll need to familiarise yourself with data structures and dictionaries in Python to make sense of the above code. Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. Keys are unique within a dictionary. To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value.

The above code comes into play after we have gone through the entire JSON file and scavenged data from all the tweets. The code block will filter the top 10 tweets based on the descending order of their occurrence, and print the results. The complete code for this tutorial is available in our GitHub repository at: **https://github.com/geekybodhi/techmadesimple-tweepy**. ■

Python's package management

Just like your Linux distribution, Python also ships with a package management system, which helps install complex packages, libraries and extensions. Python's package management system is called *Conda*, and it can be used for installing multiple versions of software packages.

Conda isn't available as a standalone application and instead ships with all Python distributions, such as Anaconda and Miniconda. Of the two distributions, Miniconda is the smaller one, and it includes *Conda* and Conda-build, and also installs Python. You can use Miniconda to install over 200 scientific packages and their dependencies with the `conda install` command.

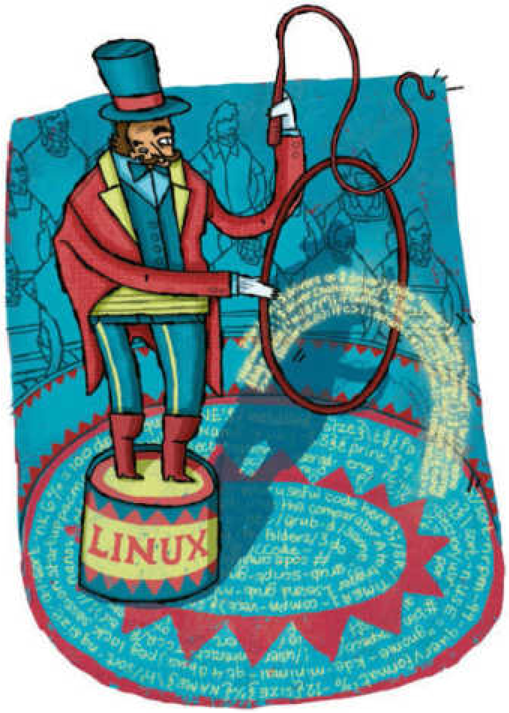
Anaconda, on the other hand, includes *Conda*, Conda-build, Python, and also over 100 packages and libraries that it installs automatically. Like Miniconda, you can install over 200 packages. To install Miniconda, grab the installer from **http://conda.pydata.org/miniconda.html** and install it with:

```
bash Miniconda3-latest-Linux-x86_64.sh
```

Then close and re-open the terminal window for the changes to take effect. Now use the `conda list` command to list the installed packages. You can now install packages such as **matplotdb** with `conda install matplotdb`.

Tkinter: Make a basic calculator

Why not use Python's default graphical toolkit to build a simple – yet easily extensible – visual calculator?



Python provides various options for developing graphical user interfaces, such as wxPython, JPython and Tkinter. Of these, Tkinter is the standard GUI library that ships by default with Python. Tkinter is the Python interface to *Tk*, the GUI toolkit for Tcl/Tk. *Tk* was developed as a GUI extension for the Tcl scripting language in the early 1990s. One of the reasons for its popularity is that it's easier to learn than other toolkits.

Tkinter provides a powerful object-oriented interface to the *Tk* GUI toolkit. To use Tkinter, you don't need to write Tcl code, because Tkinter is a set of wrappers that implement the *Tk* widgets as Python classes.

Creating a GUI application using Tkinter is an easy task. We begin by first importing the Tkinter module, which is then used to create the graphical app's main window. Next we add one or more of the supported widgets (*see box on page 143*) to the main window, before entering the main event loop to take action against each event triggered by the user. For example, the following code will display a window:

```
import tkinter as tk
window = tk.Tk()
# Insert code to add widgets
window.mainloop()
```

Quick tip

The code in this tutorial is based on vegaseat's Updated Tiny Tkinter Calculator, listed on DaniWeb.com (**http://tinyurl.com/h3eotvf**).

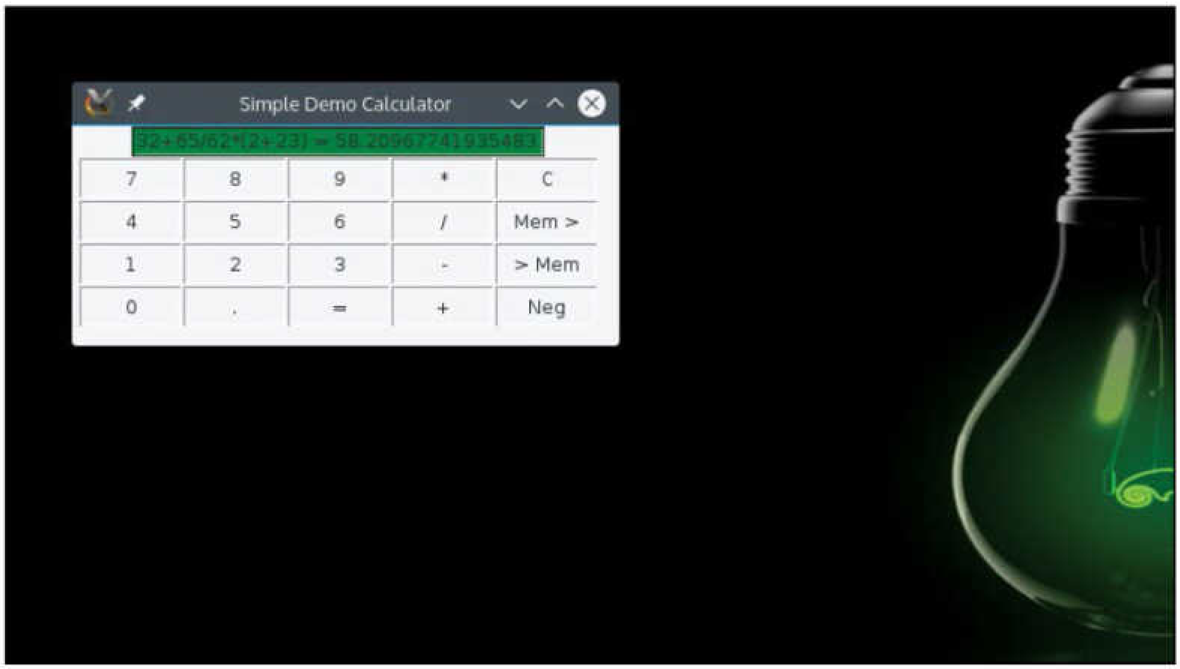
Here's a more practical example that creates a window with a title and two widgets, one of which quits the application when clicked:

```
import tkinter as tk

class App(tk.Frame):
  def __init__(self, master=None):
    tk.Frame.__init__(self, master)
    self.pack()
    self.createWidgets()

  def createWidgets(self):
    self.hi = tk.Label(self)
    self.hi["text"] = "Hello World"
    self.hi.pack(side="top")

    self.EXIT = tk.Button(self, text="EXIT", fg="red",
command=root.destroy)
    self.EXIT.pack(side="bottom")

root = tk.Tk()
app = App(master=root)
app.master.title('Sample Application')
app.master.geometry("250x70+550+150")
app.mainloop()
```

We'll explain the individual elements of the code as we build our calculator. For now, it's important to note that we've explicitly created an instance of *Tk* with `root = tk.Tk()`. Tkinter starts a Tcl/Tk interpreter behind the scenes, which then translates Tkinter commands into Tcl/Tk commands. The main window of an application and this interpreter are intrinsically linked, and both are required in order for a Tkinter application to work. Creating an instance of *Tk* initialises this interpreter and creates the root window. If you don't explicitly initialise it, one will be implicitly created when you create your first widget. While this is perfectly fine, it goes against the spirit of Python, which states that 'explicit is better than implicit'.

Graphical calculator

As you can see, it doesn't take much effort to drape a Python app with a graphical interface. We'll use the same principles to write our graphical calculator. We'll add various widgets to our calculator, including an editable display, which can be used to correct mistakes and to enter symbols and hexadecimals not on our calculator's keypad. For example, if



```
32+65/62*(2+23) = 58.20967741935483
```

❯ **Tkinter makes it relatively easy to build a basic graphical calculator in Python.**

you enter **0xAA**, it'll print the decimal equivalent, that is **170**. Similarly, the editable display can also be used to manually type in functions that aren't represented on the keypad but are defined in Python's math module, such as the logarithmic and trigonometric functions. Furthermore, our calculator will also have the ability to temporarily store and retrieve a result from its memory bank.

The complete code for the calculator is available online. Let's dissect individual elements to get a better grasp of the Tkinter library and its interactions with Python.

```
import tkinter as tk
from math import *
from functools import partial

class Calculator(tk.Tk):
  def __init__(self):
    tk.Tk.__init__(self)
```

We begin by importing the various libraries and functions for our calculator. Besides the Tkinter toolkit, we'll also need the math library to handle the calculations. The partial function from the functools module helps us write reusable code. We'll explain how we've used it later in the tutorial.

After importing the libraries, we begin defining the Calculator class. To begin with, we initialise the instance variables with the `__init__` method in the class body. This method is executed automatically when a new instance of the class is created. Python passes the instance as the first argument, and it is a convention to name it **self**. In other words, the `__init__` method is the constructor for a class in Python. The basic idea is that it is a special method, which is automatically called when an object of that class is created. So when you call `Calculator()`, Python creates an object for you, and then passes it as the first parameter to the `__init__` method.

The **self** variable and the `__init__` method are both OOP constructs. The **self** variable represents the instance of the object itself. Most object-oriented languages pass this as a hidden parameter to the methods defined on an object, but Python does not. You have to declare it explicitly. When you create an instance of the Calculator class and call its methods, it will be passed automatically. It is important to use the self parameter inside an object's method if you want to persist the value with the object.

Define the layout

The first order of business is to create the graphical interface for the calculator. In this section we define different parameters for the positioning and appearance of our »

Commonly used Tkinter widgets

The Tkinter toolkit supports over a dozen types of controls or widgets that you can use in your graphical application. Here are some of the most commonly used ones.

The Button widget is used to create buttons that can display either text or images. You can define a function for a button, which is called automatically when you click the button. The Checkbutton widget is used to display a number of options to a user as toggle buttons, and can also display images in place of text. Similarly, the Listbox widget is useful for displaying a list of items from which a user can select multiple options. The Message widget provides a multiline and non-editable object that displays texts.

Then there's the Menubutton, which creates the part of a drop-down menu that stays on the screen all the time. Every menubutton is associated with a Menu widget, which can display the choices for that menubutton when the user clicks on it. The Menu widget can create different types of menus, including pop-up, top-level and pull-down.

The Scrollbar widget provides a slide controller that is used to implement vertical scroll bars on other widgets, such as Listbox, Text and Canvas. You can also create horizontal scrollbars on Entry widgets. The Canvas widget creates an area that can be used to place graphics, text or other widgets.

All these widgets have the same syntax:

```
w = <widgetname> (parent, option1=value,
option2=value, ... )
```

Similarly, they also share several options, such as the ones to define background colour (`bg`), the type of border (`relief`), and the size of the border (`bd`). Refer to the Tkinter documentation (**http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html**) for a list of all the options supported by each of these widgets.

» calculator. Arranging widgets on the screen includes determining the size and position of the various components. Widgets can provide size and alignment information to geometry managers, but the geometry manager always has the final say on positioning and size.

```
self.title(“Simple Demo Calculator”)
self.geometry(“350x140+550+150”)
self.memory = 0
self.create_widgets()
```

While Tkinter supports three geometry managers – namely, Grid, Pack and Place – we'll use the Place manager, which is the simplest of the three and allows precise positioning of widgets within or relative to another window.

We use the `geometry` function to define the size and position of the calculator's window – it accepts a single string as the argument in the format: `width x height + xoffset + yoffset`. The real string doesn't contain any spaces, which have been added here for easier readability. You can also just set the position of the window by only defining the x and y offset coordinates.

```
def create_widgets(self):
  btn_list = [
  ‘7’,  ‘8’,  ‘9’,  ‘*’,  ‘C’,
  ‘4’,  ‘5’,  ‘6’,  ‘/’,  ‘Mem >’,
  ‘1’,  ‘2’,  ‘3’,  ‘-’,  ‘> Mem’,
  ‘0’,  ‘.’,  ‘=’,  ‘+’,  ‘Neg’ ]

r = 1
c = 0
for label in btn_list:
  tk.Button(self, text=label, width=5, relief=‘ridge’,
command=partial(self.calculate, label)).grid(row=r, column=c)
  c += 1
  if c > 4:
    c = 0
    r += 1
```

Once we've defined the look of the calculator window, the above code defines and creates the individual calculator buttons. The `create_widgets()` method contains code to create the layout for the calculator. The `btn_list` array lists the buttons you'll find in a typical calculator keypad. They are listed in the order they'll appear on the keypad.

Next we create all buttons with a `for` loop. The **r** and **c** variables are used for row and column grid values. Within the loop, we'll first create a button using the `Button` function. The Button widget is a standard Tkinter widget used to implement various kinds of buttons. Buttons can contain text or images, and you can associate a Python function or method with each button. When the button is pressed, Tkinter automatically calls that function or method.

In our code, we implement a plain button, which is pretty straightforward to use. All you have to do is to specify the button contents (text from the array, in our case) and what function or method to call when the button is pressed using the command parameter. In our code, we call the `partial` function to pass the button number. The `partial` function, which is part of the functools module, helps write reusable code. The `partial` function makes a new version of a function with one or more arguments already filled in. Furthermore, the new version of a function documents itself.

The relief style of the button widget defines the 3D effect around the boundaries of the button. By default, a button appears raised and changes to a sunken appearance when it is pressed. Because this setting doesn't suit our calculator, we've used the ridge constant, which creates a raised boundary around the individual buttons.

After creating the button, we increment the position of the column and check if we've created four columns. If we have, then we reset the **column** variable and update the **row** variable, and draw the next row of buttons. Else, we just print the label and move on to the next button. This process is repeated until the entire keypad has been drawn.

```
self.entry = tk.Entry(self, width=33, bg="green")
self.entry.grid(row=0, column=0, columnspan=5)
```

The last bit of the calculator's visual component is the editable display, which we create using the Entry widget. The Entry widget is a standard Tkinter widget used to enter or display a single line of text. The `width` option specifies the length of the display and `bg` its background colour. In the next line, the `entry.grid` method helps us place the widget in the defined cell and define its length.

Define the special keys

Our calculator is now visually complete. If you call on the Calculator class with `MyApp().mainloop()` it draws the keypad of the calculator. However, don't forget to remove the `command` parameter from the `tk.Button` function. The `command` parameter refers to the `self.calculate` function, which does the actual calculation but hasn't been defined yet. Once you've perfected the look and feel of your calculator, you can define the `calculate` function.

```
def calculate(self, key):
  if key == ‘=’:
    result = eval(self.entry.get())
    self.entry.insert(tk.END, “ = ” + str(result))
  elif key == ‘C’:
    self.entry.delete(0, tk.END)
```

In the above calculation code, we use several Entry widget methods. To add a value to the widget, we use the `insert` method, while the `get` method is used to fetch the current value. We use a combination of these to calculate and display the result whenever the equal to (=) key is pressed. The calculation is handled by the `eval` function, which parses the passed expression as a Python expression. In simpler terms, this means that the `eval` function makes a string with integers in it, a mathematical equation. The return value is the

Popular Tkinter extensions

While the Tkinter toolkit is fairly extensive, it can be extended to provide even more widgets. The most popular Tkinter extension is the tkinter.tix library, which provides several modern widgets that aren't available in the standard toolkit, such as a ComboBox and more.

Another popular toolkit is the Python megawidgets, also known as Pmw. It includes several widgets such as buttonboxes, notebooks, comboboxes, dialog windows and more. Then there's the TkZinc widget, which is very similar to the standard Canvas widget in that it can be used to implement items for displaying graphical components. However, unlike the Canvas widget, the TkZinc widget can structure the items in a hierarchy and comes with support for scaling and rotation. Furthermore, it also provides functions from OpenGL, such as anti-aliasing and more. You can find some of these widget extensions in the repositories of popular distributions. If they aren't available in your distro, grab and compile them following the straightforward instructions on their respective websites.

result of the evaluated expression. To clear the current entry, we use the `delete` method. For better control, the Entry widget also allows you to specify character positions in a number of ways. `END` corresponds to the position just after the last character in the Entry widget. We use these to clear the contents of the editable display at the top whenever the C button is pressed.

```
elif key == ' > Mem':
  self.memory = self.entry.get()
  if '=' in self.memory:
    val = self.memory.find('=')
    self.memory = self.memory[val+2:]
    self.title('Memory =' + self.memory)
  elif key == 'Mem >':
    self.entry.insert(tk.END, self.memory)
```

In this part of the code, we define the actions for saving and retrieving a calculated value from the calculator's memory banks. The current value in the buffer is the complete equation. When the button labelled '> Mem' is pressed, the code scans the equation and only saves the numbers after the equal to (=) sign. Secondly, after committing a value to the memory, it changes the title of the calculator window to reflect the contents of the memory. Conversely, when the code detects that the key labelled 'Mem >' has been pressed, it copies the number saved in its buffer to the editable display.

```
elif key == 'Neg':
  if '=' in self.entry.get():
    self.entry.delete(0, tk.END)
  try:
      if self.entry.get()[0] == '-':
        self.entry.delete(0)
      else:
        self.entry.insert(0, '-')
  except IndexError:
    pass
else:
  if '=' in self.entry.get():
    self.entry.delete(0, tk.END)
  self.entry.insert(tk.END, key)

app = Calculator()
app.mainloop()
```

The conditions in the above listed code block are a little tricky. The code at the top of this code block comes into play when the key labelled 'Neg' is pressed. If there's an equal to (=) sign in the display, pressing the key labelled 'Neg' will clear the contents of the display. If the value already contains a negative number, pressing the 'Neg' button will remove the negative sign. If, however, both the above two conditions aren't met, then the key will insert a negative sign to the beginning of the value.

This brings us to end of our code and the `Else` condition, which is triggered when none of the previously mentioned conditions are met. In this case, if there's an equal to (=) sign in the Entry widget, then that means a calculation has been completed and the result has been displayed. This means we can now clear the entry and start a new calculation as soon as a number is punched.

Add more functionality

The above code implements a functional graphical calculator. But as it is with every piece of code written for demonstration purposes, there's always scope for improvement. The good thing about Python and Tkinter is that they make it very easy to extend and improve the code. For instance, if you wish to make the code compatible with the older Python 2, you can replace the lines at the top to import the Tkinter library with the following code block:

```
try:
  import Tkinter as tk
except ImportError:
  import tkinter as tk
```

This code block will first attempt to import the Tkinter library, which is how it was referred to in Python 2. If this operation throws an ImportError, then Python will attempt to import the library with its Python 3 name.

Another weak point in the code is its use of the `eval()` function. The risk with `eval()` is well documented and has to do with the fact that it evaluates everything passed to it as regular Python code. So if the user enters a valid Python shell string into the function, it'll be executed on the server and can wreck havoc.

To get an idea of what happens, run the calculator and in the display field, instead of a number, write `__import__('os').getcwd()`. When you press the = key, the `eval()` function will execute this command on the web server and print the current working directory. The `__import__` function accepts a module name (`os` in this case) and imports it. We then use the imported module to run commands on the underlying operating system.

To prevent such exploitations of the `eval()` function, you can insert the following code while defining the `calculate` function:

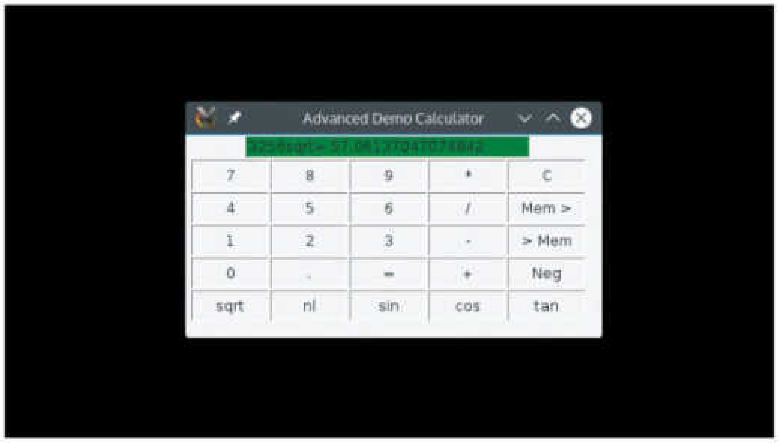
```
if '_' in self.entry.get():
  self.entry.insert(tk.END, "Nice try, schmuck! Stick to
calculations.")
```

This code is triggered when it detects an underscore (_) in the display field. Instead of passing the contents of the field to the `eval` command, the code displays a warning.

Another way to extend the calculator is to add more functions. For example, you can easily add a key to calculate the square root of a number. First add a key labelled 'sqrt' in the `btn_list` array. Then scroll down to where you define the `calculate` function and enter the following just above the final `else` condition:

```
elif key == 'sqrt':
   result = sqrt(eval(self.entry.get()))
   self.entry.insert(tk.END,"sqrt= "+str(result))
```

When the code is triggered, it'll calculate the square root of the number in the display field and print the results. You can similarly extend the calculator by implementing other maths functions as well. ■



› **Thanks to Tkinter's extensibility, you can add more functions to your calculator so it's capable of more advanced calculations.**

The best programming guide money can buy!
NEW!
CODING
MADE SIMPLE
Learn how to program the easy and fun way
LEARN HOW TO...
FREE
Digital edition of this book
See page 146 for more information
Future

GET YOUR FREE DIGITAL EDITION*

Experience this great book on your iPad or iPhone completely free!

To get your digital copy of this book simply download the free *Linux Format* app from **http://myfavm.ag/1zvbh9s** and tap Help on the bottom menu bar, then 'How do I download my reward?' and enter the voucher code **WQBA7**

*Requires an iPad or iPhone and the Apple App Store. Offer ends 18 June 2017.

```
new todolist --skip-test-unit respond_to do |format| if @task.update_attributes(params[:task]) format.html { redirect_to @task, notice: '...' } format.json { head :no_content } el
format.html { render action: "edit" } format.json { render json: @task.errors, status: :unprocessable_entity } $ bundle exec rails generate migration add_priority_to_tas
priority:integer $ bundle exec rake db:migrate $ bundle exec rake db:migrate $ bundle exec rails server validate :due_at_is_in_the_past def due_at_is_in_the_past error
add(:due_at, 'is in the past!') if due_at < Time.zone.now #!/usr/bin/en python import pygame from random import randrange MAX_STARS = 100 pygame.init() screen
pygame.display.set_mode((640, 480)) clock = pygame.time.Clock() stars = for i in range(MAX_STARS): star = [randrange(0, 639), randrange(0, 479), randrange(1, 16)] star
ppend(star) while True: clock.tick(30) for event in pygame.event.get(): if event.type == pygame.QUIT: exit(0) #!/usr/bin/perl $numstars = 100; use Time::HiRes qw(uslee
use Curses; $screen = new Curses; noecho; curs_set(0); for ($i = 0; $i < $numstars ; $i++) { $star_x[$i] = rand(80); $star_y[$i] = rand(24); $star_s[$i] = rand(4) + 1; } while (1
$screen->clear; for ($i = 0; $i < $numstars ; $i++) { $star_x[$i] -= $star_s[$i]; if ($star_x[$i] < 0) { $star_x[$i] = 80; } $screen->addch($star_y[$i], $star_x[$i], "."); } $screen->refres
usleep 50000; gem "therubyracer", "~> 0.11.4" group :development, :test do gem "rspec-rails", "~> 2.13.0" $ gem install bundler $ gem install rails --version=3.2.12 $ rben
rehash $ rails new todolist --skip-test-unit respond_to do |format| if @task.update_attributes(params[:task]) format.html { redirect_to @task, notice: '...' } format.json { hea
no_content } else format.html { render action: "edit" } format.json { render json: @task.errors, status: :unprocessable_entity } $ bundle exec rails generate migration ad
priority_to_tasks priority:integer $ bundle exec rake db:migrate $ bundle exec rake db:migrate $ bundle exec rails server validate :due_at_is_in_the_past def due_at_is_i
the_past errors.add(:due_at, 'is in the past!') if due_at < Time.zone.now #!/usr/bin/en python import pygame from random import randrange MAX_STARS = 100 pygam
init() screen = pygame.display.set_mode((640, 480)) clock = pygame.time.Clock() stars = for i in range(MAX_STARS): star = [randrange(0, 639), randrange(0, 479), randrange(
16)] stars.append(star) while True: clock.tick(30) for event in pygame.event.get(): if event.type == pygame.QUIT: exit(0) #!/usr/bin/perl $numstars = 100; use Time::HiR
qw(usleep); use Curses; $screen = new Curses; noecho; curs_set(0); for ($i = 0; $i < $numstars ; $i++) { $star_x[$i] = rand(80); $star_y[$i] = rand(24); $star_s[$i] = rand(4) +
 while (1) { $screen->clear; for ($i = 0; $i < $numstars ; $i++) { $star_x[$i] -= $star_s[$i]; if ($star_x[$i] < 0) { $star_x[$i] = 80; } $screen->addch($star_y[$i], $star_x[$i], "."
$screen->refresh; usleep 50000; gem "therubyracer", "~> 0.11.4" group :development, :test do gem "rspec-rails", "~> 2.13.0" $ gem install bundler $ gem install ra
--version=3.2.12 $ rbenv rehash $ rails new todolist --skip-test-unit respond_to do |format| if @task.update_attributes(params[:task]) format.html { redirect_to @task, notic
...' } format.json { head :no_content } else format.html { render action: "edit" } format.json { render json: @task.errors, status: :unprocessable_entity } $ bundle exec ra
generate migration add_priority_to_tasks priority:integer $ bundle exec rake db:migrate $ bundle exec rake db:migrate $ bundle exec rails server validate :due_at_is_in_th
past def due_at_is_in_the_past errors.add(:due_at, "is in the past!") if due_at < Time.zone.now #!/usr/bin/en python import pygame from random import randrange MAX
STARS = 100 pygame.init() screen = pygame.display.set_mode((640, 480)) clock = pygame.time.Clock() stars = for i in range(MAX_STARS): star = [randrange(0, 63
randrange(0, 479), randrange(1, 16)] stars.append(star) while True: clock.tick(30) for event in pygame.event.get(): if event.type == pygame.QUIT: exit(0) #!/usr/bin/p
$numstars = 100; use Time::HiRes qw(usleep); use Curses; $screen = new Curses; noecho; curs_set(0); for ($i = 0; $i < $numstars ; $i++) { $star_x[$i] = rand(80); $star_y[
= rand(24); $star_s[$i] = rand(4) + 1; } while (1) { $screen->clear; for ($i = 0; $i < $numstars ; $i++) { $star_x[$i] -= $star_s[$i]; if ($star_x[$i] < 0) { $star_x[$i] = 80; } $scree
->addch($star_y[$i], $star_x[$i], "."); } $screen->refresh; usleep 50000; gem "therubyracer", "~> 0.11.4" group :development, :test do gem "rspec-rails", "~> 2.13.0" $ gem inst
bundler $ gem install rails --version=3.2.12 $ rbenv rehash $ rails new todolist --skip-test-unit respond_to do |format| if @task.update_attributes(params[:task]) format.htm
redirect_to @task, notice: '...' } format.json { head :no_content } else format.html { render action: "edit" } format.json { render json: @task.errors, status: :unprocessable_ent
 $ bundle exec rails generate migration add_priority_to_tasks priority:integer $ bundle exec rake db:migrate $ bundle exec rake db:migrate $ bundle exec rails server valid
```

CODING MADE SIMPLE

Build essential programming skills from the ground up

Everyone can code with these plain-English guides – you'll learn core programming concepts and create code to be proud of!

helloworld.py - /home/jonni/python - Geany
File Edit Search View Document Project Build Tools Help
Symbols Documents
Functions
loop [7]
main [4]
helloworld.py
1.
2.
3.
4
5
Status
Compiler
XPS

```
#!/usr/bin/env python3
# Python intro 2016

def main():
    print('Hello world')

def loop():
    for j in range(5):
        print(j)

if __name__ == '__main__':
    main()
```

```
17:59:13: This is Geany 1.23.1.
17:59:13: Failed to load one or more session files.
17:59:13: File /home/jonni/python/helloworld.py opened(1).
18:00:58: File /home/jonni/python/helloworld.py saved.
18:01:47: File /home/jonni/python/helloworld.py saved.
```

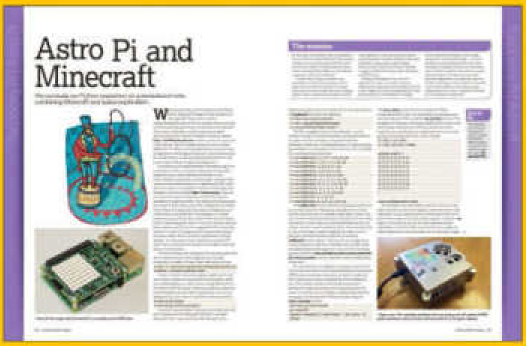
line: 2 / 13 col: 14 sel: 0 INS TAB mode: Unix (LF) encoding: UTF-8 filetype: Python

148 pages of tips, tricks and tutorials

Dozens of expert tutorials

- Pick up the basics with Python
- Teach kids code on the Raspberry Pi
- Hack Minecraft! Build a GUI! Scan Twitter!

Learn as you code with stacks of example-packed guides

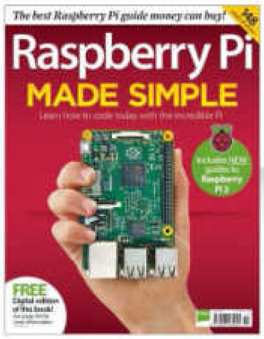


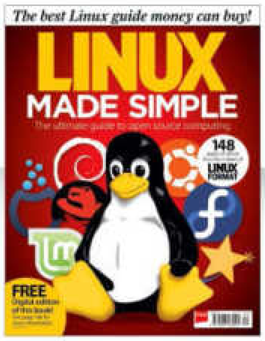
Get to grips with hacking and coding with the amazing Raspberry Pi

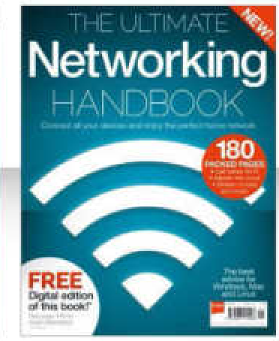
Take coding concepts further with our exciting and easy-to-follow projects

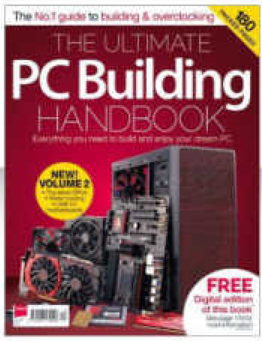
9001

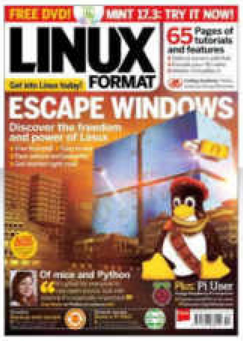
Like this? Then you'll also love...











Visit **myfavouritemagazines.co.uk** today!

9006